图形
图像
动画
专家

OpenGL

[编程与实例]

向世明 编著

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
URL: http://www.phei.com.cn

图形图像动画专家

OpenGL 编程与实例

向世明 编著

電子工業出版社

Publishing House of Electronics Industry

北京·BEIJING

内容提要

本书介绍了应用 OpenGL 开发三维图形的方法。内容包括:怎样利用 Visual C++ 5.0 MFC 中的类建立 OpenGL 应用程序的基本格式,建立和控制 2D 形状和 3D 形体、标准二次曲面、Bezier 曲线及曲面、NURBS 曲线及曲面、点光源、无限光源、聚光、基本材质、贴图材质、Mip 贴图、位图字符、文字、OpenGL 动画等,观察物体,操作像素、位图、图像,融合和反走样物体,雾化场景,利用帧缓存进行颜色屏蔽、深度控制、遮掩、全局场景反走样、景深模拟和场景对焦效果处理,利用对话框、菜单、鼠标选择物体、信息反馈实现 OpenGL 基本的交互图形程序设计。最后,本书通过 6 个实用例程说明如何将 OpenGL 的建模技术、图像处理技术和人机交互技术等 3 大技术进行结合。

本书涉及了所有的 OpenGL 核心函数和实用库函数,注重 OpenGL 的内存联系和实际运用。书中的每个工程文件模式固定且十分简洁,内容循序渐进。本书配有 CD-ROM 一张,包含了书中所有源程序和 18 个工程文件所生成的可执行文件等内容,帮助读者学习。本书适合于具有基本计算机图形学基础知识及 Visual C++ 5.0 MFC 编程基础知识的计算机软件开发人员和工程技术人员等阅读。

未经许可,不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有,翻版必究。

图书在版编目(CIP)数据

OpenGL 编程与实例/向世明编著. - 北京:电子工业出版社,1999.9
(图形图像动画专家/文宏武策划)
ISBN 7-5053-5625-9

Ⅰ.0… Ⅱ.向… Ⅲ.图形软件-软件接口.Opengl Ⅳ.TP334

中国版本图书馆 CIP 数据核字(1999)第 31908 号

丛 书 名:图形图像动画专家
书　　名:**OpenGL 编程与实例**
编 著 者:向世明
审　　者:贾志刚
责任编辑:文宏武
特约编辑:张激扬
印 刷 者:北京京安达明印刷厂
出版发行:电子工业出版社　URL:http://www.phei.com.cn
　　北京市海淀区万寿路 173 信箱　邮编 100036
经　　销:各地新华书店
开　　本:787×1092　1/16　印张:30.25　字数:774 千字
版　　次:1999 年 9 月第 1 版　1999 年 9 月第 1 次印刷
书　　号:ISBN 7-5053-5625-9/TP·2880
印　　数:5000 册　　定价:50.00 元(含光盘)

凡购买电子工业出版社的图书,如有缺页、倒页、脱页、所附磁盘或光盘有问题者,请向购买书店调换。
若书店售缺,请与本社发行部联系调换。电话　68279077

前　言

1998 年春,《泰坦尼克号》在全球成功放映,使人们首次注意到 3D 数字特技效果的新浪潮;3D Studio MAX 2.0 的推出,使 PC 机用户看到了新世纪室内动画大制作的曙光;气象服务数字化地显示了天气的变化等等。玩一玩 3D 游戏,亦或乘上网巴,我们无不体验到 3D 世界的全新感觉。我们思索:3D 产品的技术支持是什么?

3D 的技术支持是 3D 的 API。令人振奋的是,自 1992 年 7 月 SGI 公司发布了作为 3D API 的第一版 OpenGL 以来,OpenGL 经受了实践的检验,越来越获得包括 SGI、Microsoft、SUN、IBM 等在内的大型计算机企业的支持。今年,OpenGL 1.2 版已经发布,并且 Microsoft 已将其集成于 Windows NT 4.0 和 Windows 98 中。

OpenGL 集成了所有曲面造型、图形变换、光照、材质、纹理、像素操作、融合、反走样、雾化等等复杂的计算机图形学算法。应用 OpenGL 绘制图形就像我们使用 Windows GDI 的 MoveTo()函数一样方便。随着微处理器性能的不断提高和 32 位操作系统的出现,使得运用 OpenGL 开发的三维图形质量正在接近工作站的水平。

本书的特点

本书采用循序渐进的方法介绍 OpenGL 的编程技巧,同时注重 OpenGL 函数的内在联系,力求做到既介绍基本知识又提炼编程思想。本书的程序全部在 Visual C++ 5.0 MFC 环境中完成,每个例程均介绍 OpenGL 的一个主题,每个主题中又有许多知识点,每个 OpenGL 知识点均通过实例来体现。这些实例是通过视窗类(CView 类)来集成,通过菜单来观察和学习。因此,每个视窗类均有一个或多个成员变量,它们是枚举成员中的一个,其选定方法是通过响应菜单来完成。对于每一个枚举成员,均有一个视窗类成员函数与之相对应,该函数具体介绍 OpenGL 的知识点,并由 switch 语句来组织。这就是本书所采用的固定编程模式。该模式便于集中精力学习 OpenGL,而绝不是借此学习 Visual C++ 的知识。本书大量地涉及到为类添加成员变量或成员函数的操作,凡未经指明时,均针对视窗类。

考虑到辅助库函数的可移植性差,本书并不介绍 OpenGL 辅助库函数的用法。比如,辅助库函数提供了绘制圆环面的函数,但其精度不能交互地控制,因此本书并不用它,而是提供一个更好的绘制函数。在本书诸多例程中,仅一次应用了一个制作茶壶的函数。

读者对象

阅读本书并不需要较深的计算机图形学知识和 Visual C++ 的资深编程经验。但是懂得计算机图形学的基本概念和 Visual C++ MFC 编程的基本方法无疑对于理解 OpenGL 和快速运用 OpenGL 将是十分重要的。

如果读者没有这两方面的基本知识,阅读本书也没有任何障碍,因为本书不仅采用循序渐进的介绍方式,而且对于任何 Visual C++ 编程(哪怕是添加一个头文件等工作)都是严格按步骤来进行的。对于计算机图形学的基本概念,相应的地方也有必要的介绍。

为了便于学习,第 16 章的 6 个实用程序代码均被完整地列出。因而,本书的设计面向各

层次的计算机图形图像处理的专业人员和一般计算机爱好者。由于 OpenGL 具有高度的开放性、可移植性、网络透明性等特点，OpenGL 可适用于游戏开发、科学计算可视化(运动仿真、有限元分析、流体计算、医学成像、分子化学、汽车等曲面造型、地理信息、石油勘探、气候模型模拟)、虚拟现实技术、影视广告、CAI 等等领域内的广大工程技术人员和教学科研人员学习使用。

本书基于 Windows 操作系统和 Visual C++ 编程环境，所有与 OpenGL 有关的格式和程序控制均可由读者按步骤亲自编程，从而读者可领会 OpenGL 的强大功能和 OpenGL 的实质。

本书附带的 CD-ROM

必须指出，即使没有这张 CD-ROM，阅读本书和按步骤进行编程也没有任何障碍，但无疑这张 CD-ROM 必将大大地加快读者阅读的学习进程。CD-ROM 中不仅包含了本书中所有的源程序代码，而且附有 Win32 Application 样板程序等。具体内容参见附录介绍。在本书的第 3 章至第 16 章中，作者并不希望读者去先运行 CD-ROM 中对应的工程文件，而是希望读者按步骤进行编程，以从中体会。

所需环境

需要 Windows 95(98)开发环境和 Visual C++ 2.0 以上的开发工具。建议使用 Visual C++ 5.0 或 6.0。本书中的所有例程均在 Visual C++ 5.0 MFC 中完成。

主要内容

第 1 章，进入三维真彩世界。三维图形现实的市场需要为 OpenGL 提供了应用的大舞台。

第 2 章，OpenGL 概述。本章的主要内容有：OpenGL 的发展简史、OpenGL 的特点、OpenGL 工作结构和功能、OpenGL 的工作原理、图形操作步骤、OpenGL 函数库、OpenGL 结构、OpenGL 数据类型和 OpenGL 句法等。

第 3 章，OpenGL 极小化工作。就像我们应用 GDI 函数之前要初始化设备描述表对象一样，为使 OpenGL 获得硬件支持，必须建立 OpenGL 的像素格式、当前化着色描述表。本章介绍了如何利用 Visual C++ MFC 基本类和 Windows 消息来解决这一问题。

第 4 章，绘制二维图形。二维图形包括点、线、面。本章介绍了如何生成二维图形及对二维图形的控制方法，比如，如何生成点画线、如何控制线宽、如何定义一个多边形等等基本造型知识。作为扩展，本章还介绍了如何构造三维形体的基本方法，并由二维的面拼接了一个盒子。

第 5 章，OpenGL 变换。场景是由许多物体有序组合而成。将一个三维场景映射成计算机屏幕区域上的一个二维图像需要一系列的变换，OpenGL 集成了这些算法，使编程者可以模拟架设相机、选择场景、调整拍摄角度、剪切场景及投影成像等照相过程。

第 6 章，显示列表。OpenGL 可以通过所谓的显示列表来优化被多次调用的代码。只要后继时间允许，OpenGL 就会执行它们，这对提高应用程序(特别是网络程序)的整体性能将是十分重要的。

第 7 章,模拟真实世界。光照和材质是生成真实感三维图形的基本要素。本章讨论材质定义、点光源、平行光源和聚光的生成与控制,光的模拟,颜色缓存、深度缓存的操作与应用,着色模式的选择与应用,以及颜色、材质和光三者的相互作用,多光源的相互作用等。通过光照一个盒子和球来应用这些基本知识,并采用对话框的形式实时地控制颜色、颜色屏蔽、光照、材质等参数。通过本章,还可以学习制作 OpenGL 交互程序的基本方法。

第 8 章,OpenGL 动画。OpenGL 动画采用双缓存机制。事实上,生成 OpenGL 动画的机制是十分简单的,主要应明确运动对象的运动学条件。

第 9 章,OpenGL 图像处理基础。本章介绍如何操作像素、位图和图像,如何生成位图字符和输出 Windows 已有字库的字体。

第 10 章,纹理贴图。纹理为真实造型提供了一个灵活的工具。本章不仅介绍纹理的生成、控制与映射等基本知识,而且还将制作一个专门处理位图的类,通过该类,可以轻松地将一个任意给定的 Windows 24 位位图贴到物体的表面,介绍如何将纹理贴图到曲面物体上,如何运用 Mip 贴图等高级纹理贴图技术。

第 11 章,增强图像效果。它们均属于 OpenGL 高级图像生成技术。本章通过实例介绍了采用融合将多个物体颜色进行混合生成新物体,制作透明物体,对点、线和面进行反走样,以及通过雾化模拟真实自然界的方法。

第 12 章,实用库函数。本章介绍了 OpenGL 实用库函数。重点介绍标准二次曲面的生成、图像操作和自动 Mip 贴图方法。

第 13 章,曲线和曲面。Bezier 曲线和曲面,NURBS 曲线和曲面在工程中有着广泛的应用。本章介绍这两类曲线和曲面、纹理曲面等的生成与控制,还介绍了如何制作有洞曲面(连通曲面)的方法。

第 14 章,高级制作技术。主要内容有:模板缓存的操作与应用,累积缓存的操作与应用。本章通过模板缓存制作了一块正方形掩膜并将一个球绘制在该掩膜上,利用模板缓存实现对两球交集的制作;利用累积缓存进行场景全局反走样、景深模拟和对焦效果处理。

第 15 章,交互技术。OpenGL 的 3 个基本模式即绘制、选择与反馈模式构成了利用 OpenGL 开发交互式三维图形的基础。主要内容有:选择模式的建立与操作步骤、反馈模式的建立与应用。本章利用选择模式,将绘图区等分成 4 个区域,当鼠标单击一个区域时,启动选择机制并改变该块区域的背景颜色;利用反馈模式绘制着色了两个点,并将这两个点的坐标和颜色值打印在绘图区指定的位置。

第 16 章,OpenGL 全面制作技术。本章集中了 6 个实用例程,通过它们重点说明了 OpenGL 的 3 大技术:建立模型技术、图像处理技术和人机交互技术,并将这些技术结合起来。第一个例程说明了建立数学模型是解决问题的一个好途径。该例程演示了兔子和乌龟的追逐轨迹以及如何制作分形图形。物体建模技术是最核心的技术也是最难处理的技术。因此,第二和第三个例程说明建立物体几何模型的理论方法,用实例说明怎样去组织好所需数据(几何顶点、颜色、法线和纹理坐标等)及数据结构并与所需要目标结合起来,怎样为人机交互提供接口;怎样实现由数据驱动的动画。第二个例程制作一面飘扬的旗帜,第三个例程则显示一个滚动的西瓜。第四个例程说明怎样将画面着色到内存位图设备,说明当指定画面中所需要的子区域时怎样将其输出到外部文件,说明怎样将简单物体(本例程中由一个线框球、几个圆圈构成)组合起来生成一个艺术处理的画面。第五个例程说明了物体阴影的制作方法。重点制作铅笔及其阴影。“铅笔”作为一类物体的总称,以类的形式给出。本例程还提供了另外两个类,

一个处理颜色，一个处理向量的各种运算、点绕轴和任意向量的旋转、点的光透点计算、光的反射方向计算等等。最后铅笔可以进行各种几何变换(封装于类中)，因而三个类不经修改可以直接移植应用。最后一个例程说明牛头刨床急回机构的运动仿真。通过该例程说明较一般的基于运动学的运动观察方法和程序设计思想。

参加本书编写工作的同志还有：华中理工大学的邓爱平、杨克升、曹淮、张玲、王峰、史美康、吴诚德，武汉水利电力大学的王考等。在此作者再一次向他们表示深深的谢意。

作者要深深地感谢华中理工大学的罗晋华和黄飞，感谢他们珍贵的指导、诚挚的关心和硬件等条件的支持；感谢华中理工大学的邓爱平，感谢她富有成效的鼓励、全力的支持和所提供的算法；感谢深圳中信公司的张佳鑫、梁三光，感谢他们的指导和所提供的资料。在此作者还要感谢重庆建筑大学的李开海、重庆邮电大学的潘显彬、重庆工业管理学院的张朝晖、重庆大学的钟佑明，感谢他们的鼓励和支持；感谢华中理工大学的史美康和王峰，感谢他们对本书程序的第一次验证和所提供的宝贵建议。

在此，作者还要感谢电子工业出版社的文宏武先生，感谢他所提供的珍贵思路。总之，如果没有这一切，这本书是不可能写成和出版的。

作者

1999 年 4 月于武昌

目　录

第 1 章　走进三维真彩世界 …… (1)
1.1　从《泰坦尼克号》说起 …… (1)
1.2　遨游虚拟世界 …… (2)
1.3　科学计算的可视化技术 …… (3)
1.4　对 3D Studio MAX 的思考 …… (4)
1.5　3D 图形加速卡 …… (4)
第 2 章　OpenGL 概述 …… (6)
2.1　OpenGL 是什么 …… (6)
2.2　OpenGL 的工作结构 …… (7)
2.3　OpenGL 能做什么 …… (7)
2.4　OpenGL 是怎样工作的 …… (9)
2.5　OpenGL 图形操作步骤 …… (10)
2.6　OpenGL 函数及结构 …… (11)
2.7　OpenGL 数据类型 …… (12)
2.8　OpenGL 句法 …… (13)
2.9　OpenGL 状态变量 …… (14)
2.10　齐次坐标表示 …… (21)
第 3 章　极小化 OpenGL …… (22)
3.1　OpenGL 像素格式 …… (22)
3.2　着色描述表 …… (24)
3.3　设置像素格式 …… (24)
3.3.1　填充 PIXELFORMATDESCRIPTOR 结构 …… (24)
3.3.2　设置像素格式 …… (25)
3.3.3　测试像素格式 …… (26)
3.4　创建着色描述表 …… (27)
3.5　极小化 OpenGL …… (30)
3.5.1　创建用户界面 …… (31)
3.5.2　极小化 OpenGL …… (32)
3.6　OpenGL 颜色 …… (38)
3.7　程序清单 …… (40)
第 4 章　绘制二维图形 …… (49)
4.1　绘制点 …… (49)
4.2　绘制直线 …… (51)
4.2.1　绘制直线 …… (51)
4.2.2　线型 …… (52)
4.2.3　绘制折线 …… (54)
4.2.4　绘制闭合折线 …… (55)

4.3 绘制多边形 …… (56)
4.3.1 绘制多边形 …… (56)
4.3.2 多边形填充模式 …… (59)
4.4 绘制三角形 …… (61)
4.4.1 绘制三角形 …… (61)
4.4.2 绘制三角形片 …… (62)
4.4.3 绘制三角形扇 …… (64)
4.5 绘制四边形 …… (66)
4.5.1 绘制四边形 …… (66)
4.5.2 绘制四边形片 …… (66)
4.6 边的可见性 …… (68)
4.7 由面创建三维形体 …… (70)
4.8 建立应用程序 TwoDShape …… (72)
4.9 TwoDShape 应用程序是如何运行的 …… (80)
第 5 章 OpenGL 变换 …… (82)
5.1 建立 GLTrans 工程 …… (82)
5.2 投影变换 …… (85)
5.2.1 平行投影 …… (85)
5.2.2 透视投影 …… (86)
5.3 几何变换 …… (88)
5.3.1 一个问题 …… (88)
5.3.2 平移变换 …… (88)
5.3.3 平移盒子 …… (90)
5.3.4 旋转变换 …… (94)
5.3.5 旋转盒子 …… (94)
5.3.6 缩放变换 …… (95)
5.3.7 放大盒子 …… (95)
5.4 视口变换 …… (96)
5.5 剪切变换 …… (97)
5.5.1 剪切变换 …… (97)
5.5.2 剪切盒子 …… (98)
5.6 OpenGL 坐标系 …… (98)
5.7 集成三维物体 …… (99)
5.7.1 认识平移 …… (100)
5.7.2 认识旋转和缩放 …… (101)
5.7.3 集成 8 个盒子 …… (101)
5.8 矩阵堆栈 …… (103)
5.8.1 矩阵堆栈的作用 …… (103)
5.8.2 绘制一辆卡车的平面模型 …… (104)
5.9 矩阵操作 …… (107)

第 6 章　显示列表 …… (110)
6.1　创建和执行显示列表 …… (110)
6.2　管理显示列表 …… (113)
第 7 章　模拟真实世界 …… (117)
7.1　生成真实图形 …… (117)
7.2　OpenGL 的颜色模型 …… (117)
7.3　建立 Light 工程 …… (119)
7.4　OpenGL 光照 …… (125)
7.4.1　光的类型 …… (126)
7.4.2　定义光的属性 …… (126)
7.4.3　光照盒子和球 …… (127)
7.5　物体的材质 …… (132)
7.5.1　定义材质 …… (133)
7.5.2　辐射光 …… (134)
7.5.3　材质和光的相互作用 …… (134)
7.5.4　制作材质 …… (134)
7.6　盒子就是盒子——法线的作用 …… (140)
7.7　明暗着色处理 …… (144)
7.7.1　明暗着色处理 …… (144)
7.7.2　平滑盒子 …… (146)
7.8　深度测试 …… (149)
7.9　光照模型 …… (151)
7.9.1　全局环境光 …… (151)
7.9.2　镜面高光 …… (151)
7.9.3　双面光照 …… (152)
7.10　聚光 …… (152)
7.10.1　定义聚光 …… (152)
7.10.2　用聚光照射物体 …… (153)
7.11　创建逻辑调色盘 …… (162)
7.11.1　逻辑调色盘 …… (162)
7.11.2　创建逻辑调色盘 …… (163)
7.12　应用程序 Light 使用导示 …… (169)
第 8 章　OpenGL 动画 …… (172)
8.1　动画生成技术 …… (172)
8.2　OpenGL 动画工作原理 …… (172)
8.3　让盒子动起来 …… (173)
第 9 章　OpenGL 图像处理基础 …… (180)
9.1　位图和图像 …… (180)
9.1.1　读取图像 …… (181)
9.1.2　绘制图像 …… (181)

9.1.3 当前光栅位置 …… (182)
9.1.4 拷贝图像 …… (182)
9.1.5 缩放图像 …… (182)
9.1.6 图像操作工程 Image …… (182)
9.2 位图字符和文本 …… (184)
9.2.1 生成位图字符 …… (185)
9.2.2 生成文本 …… (186)
第 10 章 纹理贴图 …… (188)
10.1 纹理贴图技术 …… (188)
10.2 OpenGL 纹理贴图 …… (189)
10.2.1 定义纹理 …… (190)
10.2.2 控制纹理 …… (190)
10.2.3 说明纹理贴图方式 …… (191)
10.2.4 定义纹理坐标 …… (191)
10.2.5 应用纹理贴图技术 …… (192)
10.3 位图纹理 …… (196)
10.3.1 设备无关位图(DIB)的数据结构 …… (196)
10.3.2 创建位图类 …… (198)
10.3.3 将位图贴到物体上 …… (203)
10.4 纹理坐标的自动生成 …… (206)
10.5 Mip 贴图 …… (209)
第 11 章 增强图像效果 …… (216)
11.1 OpenGL 融合 …… (216)
11.1.1 OpenGL 融合操作 …… (216)
11.1.2 融合四边形 …… (218)
11.1.3 制作透明玻璃 …… (222)
11.2 反走样 …… (226)
11.2.1 反走样思想 …… (226)
11.2.2 OpenGL 反走样 …… (226)
11.2.3 反走样的线框球和三角形 …… (228)
11.3 OpenGL 雾化 …… (232)
11.3.1 确定雾的属性 …… (233)
11.3.2 雾化场景 …… (234)
第 12 章 实用库函数 …… (238)
12.1 标准二次曲面的生成 …… (238)
12.1.1 创建二次曲面对象 …… (238)
12.1.2 绘制二次曲面 …… (240)
12.2 图像处理 …… (244)
12.3 自动 Mip 贴图 …… (246)
第 13 章 曲线和曲面 …… (252)

13.1 概述 …… (252)
13.2 Bezier 曲线 …… (252)
13.2.1 Bezier 曲线 …… (252)
13.2.2 绘制一条拟正弦曲线 …… (254)
13.3 NURBS 曲线 …… (257)
13.3.1 NURBS 曲线的生成 …… (257)
13.3.2 绘制一条 NURBS 曲线 …… (260)
13.4 Bezier 曲面 …… (262)
13.4.1 Bezier 曲面的生成 …… (262)
13.4.2 绘制网格曲面 …… (262)
13.4.3 绘制一个填充曲面 …… (265)
13.5 将位图贴到曲面上 …… (268)
13.6 NURBS 曲面 …… (271)
13.7 曲面的剪切 …… (276)
第 14 章 高级制作技术 …… (280)
14.1 模板缓存及其应用 …… (280)
14.1.1 模板缓存的操作 …… (280)
14.1.2 模板缓存的应用 …… (281)
14.2 累积缓存及其应用 …… (286)
14.2.1 累积缓存的操作 …… (287)
14.2.2 全局反走样 …… (287)
14.2.3 景深模拟 …… (293)
第 15 章 交互技术 …… (297)
15.1 物体的选择 …… (297)
15.1.1 建立名称堆栈 …… (297)
15.1.2 拾取物体 …… (298)
15.1.3 选择模式的应用 …… (298)
15.2 信息反馈 …… (308)
15.2.1 反馈模式 …… (308)
15.2.2 反馈模式的应用 …… (309)
第 16 章 制作技术与应用实例 …… (314)
16.1 建立数学模型是解决问题的重要途径 …… (314)
16.1.1 解决问题的方案 …… (314)
16.1.2 乌龟和兔子的追逐、分形 …… (315)
16.1.3 程序清单 …… (317)
16.2 数据的组织:飘动的旗帜 …… (328)
16.2.1 解决问题的方案 …… (328)
16.2.2 飘动的旗帜 …… (329)
16.2.3 程序清单 …… (331)
16.3 数据的组织:滚动的西瓜 …… (349)

16.3.1 解决问题的方案 …… (349)
16.3.2 滚动的西瓜 …… (349)
16.3.3 程序清单 …… (352)
16.4 将画面输出到外部文件 …… (367)
16.4.1 将画面着色到位图设备 …… (368)
16.4.2 有选择地输出画面 …… (372)
16.4.3 程序清单 …… (375)
16.5 制作阴影 …… (390)
16.5.1 解决问题的途径 …… (390)
16.5.2 绘制铅笔及其阴影 …… (391)
16.5.3 程序清单 …… (393)
16.6 牛头刨床急回机构的运动仿真 …… (437)
16.6.1 程序设计思想 …… (437)
16.6.2 运动仿真的实现 …… (439)
16.6.3 程序清单 …… (441)
附录 配书 CD-ROM 内容介绍 …… (470)

第1章　走进三维真彩世界

1.1　从《泰坦尼克号》说起

当前的时代是信息化、数字化的时代。数字技术的广泛应用和产业化将在人类历史上引起一场新的革命。近年来，随着计算机软硬件、超大规模集成电路、大容量光盘存储器、数学信号处理技术以及高速数字通信网络的发展，出现了数字视盘机、数字电视、数码照相机、数字电话、数字式直接印刷机等一系列基于数字技术、革命性的新产品。这些新产品在高保真、高清晰度、超大规模的存储容量、灵活的交互性、计算机信息技术处理的便利性和在网上高速传输等方面的优势，大大开阔了人们的眼界，使人们看到了未来数字化时代的曙光和发展趋势，加速了世界各国发展数字技术的步伐。在数字化革命的大潮中，计算机产生图像(CGI)即数字特技效果技术作为数字化技术的一个重要分支正在加速发展。

1998年3月23日晚，电影《泰坦尼克号》于加州好莱坞电影学院奥斯卡受奖大会上获得了11项大奖。其中的"最佳视觉效果"奖完全是基于计算机图形学发展起来的计算机数字艺术的贡献。

当观众观看《泰坦尼克号》中轮船的宏伟场景时，不仅看到了用摄像机拍摄的图像，看到了上万亿字节数据组成的数字图像，而且也看到了由计算机合成制作的一幅幅美妙绝顶的画面。我们不禁要问这是怎么完成的？比如，影片开始时，Rose面前呈现的模拟泰坦尼克号沉没过程：模拟船如何断裂，船头如何扎入海底，船尾沉入海底的过程，轮船从水面消失的动画；在上层甲板上乘客目睹的漂过轮船边的冰山形状；Jack和Rose向外看，目睹船尾和冰山相撞的镜头；当Jack和Rose爬上船尾舵楼时，一些狂暴的数字人在他们背后的上层甲板上来回奔跑的镜头等等。

Microsoft公司的三维动画创作系统SoftImage被重点应用在这部影片的动画制作上，通过摄像机跟踪后，在电脑中确定实拍镜头中物体的位置和摄像机的位置，然后把所获得的数据用于海洋、轮船、人物、场景的绘制。这些数据使得由三维动画制作的乘客的脚能够接触到甲板，行走在走廊上或依靠在栏杆上；由电脑动画制作的海水能够冲刷船体；烟雾能够从烟筒中冒出。这些工作，需要来自于以不同规格建立的模型的运动参数，并进行运动匹配。海洋的制作数据则来自于已建立的一系列用于时辰、风速、海浪起伏、太阳位置及反射参数，使之接近或用于创建一个完整的数字化海洋。海鸥、旗子、鱼、天空中的星星、撞船时从船上落下的碎片、海水与轮船之间溅起的浪花、绳子、烟雾、波动的帆船、冰山、甲板上的冰块、撞碎的玻璃等除人以外的场景均由计算机制作而成。没有人愿意模拟从230英尺高度跳入大海。当你看到船尾开始倾斜到90度，上千人绝望地惊叫着逃命的场景时，能想到其中85%的动画都是由计算机完成的吗？

《泰坦尼克号》是艺术和计算机数字艺术完美结合的杰作，它使人们首次注意到数字特技效果的新浪潮。作为一名图形软件开发与应用工作者，除了惊叹SoftImage以及INFERNO(在《泰坦尼克号》中用于制作影视后期特技效果)等软件的能力外，更应该去思考它们的底层

是如何实现的。可以肯定，OpenGL(Open Graphics Libaray)正是可以完成这项任务的底层工具。同时，还可以肯定，OpenGL 必将在 CGI 技术中发挥重大的作用。

1.2 遨游虚拟世界

谈到虚拟现实(Virtual Reality)，我们便联想到 LawnMover Man 、Johnny Mnemonic 及 Titanic 等电影中的画面。由此而获得的对虚拟世界的认识，使我们对其产生了极大的兴趣。

虚拟现实技术是一门由应用驱动的涉及众多学科的新的实用技术，是集先进的计算机技术、传感与测量技术、微电子技术等为一体的综合集成技术。在计算机技术中，虚拟现实技术的发展又依赖于人工智能、图形学、网络、面向对象、Client/Serve、人机交互和高性能计算机技术。该技术的目的是以三维图形为主，从视觉感知、听觉感知角度出发创造出一个让人身临其境的虚拟世界。

远程操作是早期虚拟现实系统工作的动力，美国 Ft.Pierce 海洋学研究所的 Anlhy Yong 及 Donna Kocak 开发出一个高强度激光探测器的远程可视系统。该系统产生水下机器人前面物体的三维图像，这种几何信息可用于指导远程手臂操作。NASA 的国家超级计算机应用中心长期致力于复杂现象的可视化研究。加拿大 Albert 大学为艺术家研制了暑期虚拟环境工作站。在教育培训上，网络虚拟环境可进行游戏、解决各种数学问题。虚拟现实技术甚至可以用来虚拟奥运会。比如，西班牙在 SGI 工作站上做的多用户 VR(Virtual Reality)项目以奥运会体育竞技作为其研究重点，以数字化虚拟方式制作 1992 年巴塞罗那奥运会的足球决赛 4 人电影剧本。

虚拟现实技术的一个核心问题之一是视景的生成。视景的生成首先应生成视景的内部条件，即，建立视景生成的过程和对象的几何模型。接着要完成视景的基本内容表示，比如图形学表示、图像表示、运动图像表示。然后要考虑视景的高层内部表示，比如，三维物体的几何模型、物体的结构层次表示等。最后完成真实视景的生成。OpenGL 正是一种支持视景生成的程序设计语言。

令我们感到十分高兴的是，虚拟现实技术离我们越来越近了。虚拟世界在 Internet 上不断发展壮大。VRML(Virtual Reality Modeling Language)已将三维世界带入了 Web。VRML 对每个人都有用，它是建立和体验一个新世界的通讯媒介，可以模拟现实世界的特征。VRML 浏览器可以带你浏览 web 上的 3D 实境。VRML 生成器能让你生成自己的虚拟世界，供别人寻访。这样人们能够参观你的工厂，访问你的高尔夫球场，在电子咖啡屋和你会谈。VRML 既可以用来建立真实世界场景的模型，也可以建立虚拟的二维、三维世界，就像许多游戏中那样。VRML 也可以用来将抽象的东西形象化。Microsoft 已经采纳了 VRML 2.0 标准，并已利用 DimensionX 基于 Java 的 Liquid Reality 来生成用于 VRML 2.0 浏览的 ActiveX 组件。可在 Windows NT 4.0 上找到 Microsoft 的优化 OpenGL 1.1。Windows 98 已实现了对 OpenGL 的支持。

在众多的 VRML 浏览器中，V.Realm Browser(免费)是一个强大的，基于 OpenGL 的浏览器。V.Realm 的寻访控制在外表和感觉上仍然具有以前模拟器特点的痕迹。V.Realm 有大量的运行参数，从静止和动态模式的着色方式到地心引力和碰撞检测，不一而足。它使用 OpenGL 加速引擎。目前只有 OpenGL 能够提供如此令人印象深刻的视觉效果。如果不借助 OpenGL，使用 V.Realm 将消耗很多时间。WebSpace Navigator(免费)来自于 Template Graph-

ics Software,是第一个可在 Web 上查看和寻访 3D 实境的浏览器。WebSpace 的寻访控制仍然是最好的。由于使用 OpenGL 作为 API 达到了非常优秀的性能和非常高质量的着色视觉效果。这是非 OpenGL 产品需要努力追赶才能达到的性能和质量。WebSpace 无懈可击地对材质进行着色,并且支持 OpenGL 光照模型,具有镜面高光、透明度、辐射和发光效果等等。GLview(非免费)是供图形专业人员使用的浏览器。这些图形专业人员在将应用程序发布之前,需要将 OpenGL 图形流水线细调。

可以肯定地说,OpenGL 正将人们带入三维世界,且随着高新尖科技发展对图形、图像处理的实际需要,OpenGL 必将发挥更加重要的作用。

1.3 科学计算的可视化技术

科学计算可视化(Visualization in Scientific Computing)是 1987 年由 B.H.Mccormick 等为美国国家科学基金会所写的一份报告(B.H.Mccormick.. Visualization in Scientific Computing . Computer Graphics,1987,Vol.6) 中首次提出的。短短 10 年,科学计算可视化发展十分迅速,并已成为计算机图形学的一个十分热门的研究方向,在很多领域,比如,有限元分析、流体计算、医学图像、分子化学等领域有着广泛的应用。

科学计算可视化的目标是把由数值计算或实验获得的大量数据按照其自身的物理背景进行有机地结合,用图像的方式来展示数据所表现的内容及其相互关系,便于把握过程的整体演进,发现其内在规律,丰富科学研究的途径,缩短研究周期。在科学计算可视化技术中,算法的研究是十分重要而又十分基础的。与计算机可视化技术的另一研究方向——运动仿真(Athletics Simulation)一样,面临着如何从数据集构造物理模型,如何建立可视化模型以及绘制技术的实现方法等重大问题。可视化技术首先要将抽象数据表示转化成图形或图像图元表示。这些图元一般为点(Piont,Pixel)、线(Line,Polyline,Spline)、面(Mesh,NURBS Surface)、图像(Pixel,Bitmap)、体素(Voxel)等等,由这些图元构成可视化模型。可视化模型的绘制技术是在计算机图形学的基础上,对几何元素的共性进行抽取,实现数据的图元化、图元的参数化、图元的属性化、图元属性的参数化。图元的参数化包括几何造型的参数控制、光照模型的参数控制、视点(View Point)的参数控制、投影类型(Orthographic,Pespective)、视觉体(Frustum)的大小等。图元属性则包括颜色(Color)、透明性(Opacity)、材料的光学性能(Ambient,Diffuse,HightLight,Specular,Shininess)及纹理(Texture)等。图元属性参数与其他参数一起,构成可交互的可视化。可视化绘制技术中面绘制技术是最基本的。它以点、线、面及其表示的几何形体如多边形、球面、柱面等为图元。而体绘制技术则是从体元直接绘制出最后的显示图像,不通过中间曲面表示。

不同的数据场所采用的可视化技术也不尽相同,一般是按切片的思想进行降维处理。以体数据场为例,它们分布在研究系统的三维空间,并且随时间的变化而变化,如结构的应力场、人体的温度场等。体数据场的可视化方法就是要把数据映射到几何形状的属性上。因研究的需要,这些时间和空间相关的体数据需要在可视化中显示出来,同时也需要将不同变量类型的数据在显示中分离出来。

可以肯定,由于 OpenGL 是处理专用图形硬件的软件接口,支持可视化实现的语言,必将在计算机可视化技术中发挥其独特的优势。

1.4 对 3D Studio MAX 的思考

强有力的关键帧动画制作软件 3D Studio MAX 是基于 OpenGL 标准的(但它所用的 API 为 Autodesk 公司的 Heidi)。该软件在微机上制作的三维场景及动画能够与图形工作站制作的三维场景及动画相媲美。如果读者使用过这个软件,请从图形学的角度思考下面的基本问题。它的点、线、面是如何着色的,球怎么就是球,立方体怎么就是立方体,等等;Bend、Taper、Bever、Twist、Extrude、Lathe 等等变形功能是如何实现的;放样是怎么实现的;光照是怎么实现的;可不可以自己生成标准材质和贴图材质;材质编辑器中双面材质、混合材质、次物体材质等是怎样制成的。其堆栈修改功能给高级三维造型带来了很大的灵活性,它们是怎么对建模进行控制的。运动面板中的诸多运动控制参数是如何控制运动的;动画到底是怎么实现的。View、World、Parent、Local 等坐标系统是怎样进行变换的;物体的变换是怎么实现的。环境效果是怎么实现的。等等。

3D Studio MAX R2 的问世,更使得该软件成为各行各业辅助设计中不可缺少的工具。我们完全可以说它是一个全新设计的超世纪的三维动画及视觉设计软件。在模型制作方面,可以应用 NURBS 顶点及曲线来自由创建复杂的曲面,再也没有点数一定要相同的限制;在动态模拟方面,可以对空气阻力、摩擦力进行参数控制;在分子运动方面,将分子转变为几何物体或指定物体,创造出逼真的爆炸碎片;在电光特效方面,可制作上百种光及镜头相关的特效,如物体发光等;在着色品质方面,可指定场景内的某些物体,以光迹法着色。3D Studio MAX R2 支持 OpenGL 图形加速的硬件。

以上所列问题及其功能,借助 OpenGL 及图形生成算法,都能一一解决(注意,3D Studio MAX 中的很多用语和 OpenGL 是一致的)。但是必须注意,现在的 3D Studio MAX 的动画是关键帧动画,并不是人机交互的,若借助于 Windows 编程环境及 OpenGL 可以做到这一点。

1.5 3D 图形加速卡

迄今为止,微处理器的性能几乎每 18 个月翻一番,但与 3D 图形处理芯片的发展速度相比,它就显得黯然失色了。图形处理芯片的性能每 6 个月左右就翻一番,这使得高档的 3D 图形加速卡层出不穷。3D 图形处理加速卡的市场竞争也十分激烈。

专业级 3D 图形加速卡大多采用专门的 3D 图形加速芯片,显示卡有 8M、16M 甚至更大的显示内存。一个或一组 3D 图形加速芯片,在 Windows NT 及 OpenGL 环境下具有异常强大的 3D 图形加速功能,广泛应用于 AutoCAD、3D Studio MAX 及游戏开发中。

OpenGL 作为一个 3D 的 API,尽管在 Windows 95(98)平台、640′480 分辨率、16 位增强色模式 、奔腾 133、16M 内存、普通显卡等软件环境和硬件支持下,可以进行绘画,但其绘画速度必然会受到严重影响。如果作为 3D 图形应用开发(例如游戏开发),配置支持 OpenGL 的 3D 图形加速卡将是必要的。这不仅节省大量的着色时间,更重要的是,可以应用高质量的图形性能(如:点抽样,透视校正,双线性映射,雾,镜面反射及 Mip 映射等)来开发出逼真的三维场景。下面简单介绍几种支持 OpenGL 的 3D 图形加速卡。

中国江西科环公司的 GraphExpress MD 三维图形加速卡,使用 GLINT Delta 作为几何流水线加速处理器,一块 GLINT MX 作为渲染处理器。其存储器上有 8M VRAM(Video RAM)

及 8M EDO DRAM(Extended Data Out DRAM) ,32 位 Z 缓存 (Z Buffer),最大可达 32MB 帧缓存和 48 位 Z 缓存,最大图像精度 2048 × 2048。GraphExpress MD 支持 OpenGL,它固化了 OpenGL 的所有函数,提供 100% 的 OpenGL 1.1 硬件加速,每秒可处理 100 万个多边形的纹理映射,使用 8 列 VRAM 块填充,每秒可进行 4GB 的像素填充,支持双线性(Bilinear)和三线性(Trilinear)过滤。用于专业级图形工作站,价格较高。

美国 Diamoad 公司的三维图形加速卡 Fire GL 3000 也使用 GLINT Delta 作为图形加速处理器,但其渲染处理器为 GLINT 500TX。有 8M VRAM 帧缓存,8M EDO DRAM 的 32 位缓存。最大图像精度为 2048 × 2048。Fire GL 3000 支持 OpenGL,实现 100% 的 OpenGL 硬件加速,每秒可处理 50 万个多边形纹理映射,支持双线性及三线性过滤。用于专业级图形工作站,价格较高。

在电脑图形处理和数字通信领域享誉世界的德国厂商艾尔莎(ELSA)新近推出一款新图形加速卡 Gloria Synergy,是一种专门用于 CAD 应用领域的图形加速卡。Gloria Synergy 采用 AGP 总线、3D labs 的 Permedia2 处理器和 8MB SGRAM。Gloria Synergy 能在硬件中提供双线性纹理映射,支持 OpenGL。

加拿大 Matro 公司的 MGA Mystique 加速卡则采用第五代 64 位图形加速芯片 1064SG 于 MMX 系统中,采用比 EDO DAM 快几倍的 SGRAM(Synchronous Graphics RAM)缓存。真彩模式下分辨率可达 1280 × 1024;支持 OpenGL。Matro 公司的商品化产品 MGA G200 系列则在 1000 元左右,主要用于 PC 机。

目前,市场上有性能各异的 3D 图形加速卡。在利用 OpenGL 进行图形图像开发的过程中,应视软件开发的规模及自身的经济实力等因素决定是否配置支持 OpenGL 的高档的 3D 图形加速卡。

应该注意的是,由于技术的进步及市场的竞争,3D 图形加速卡新品倍出,当读者读到本书时,也许上面所述诸产品有的已被更高性能的产品所取代。

第 2 章　OpenGL 概述

2.1　OpenGL 是什么

长期以来，人们致力于计算机三维图形技术的研究。由于生成真实感图形需要较深的数学知识(三角学、线性代数、微分几何等)，今天的许多三维图形开发工具已经隐藏了这些细节，使编程人员仅仅只需要考虑怎样创作自己的图像。OpenGL 就是这样的一个开发工具。

OpenGL 最初是 SGI 公司为其图形工作站开发的可以独立于窗口操作系统和硬件环境的图形开发环境，其目的是将用户从具体的硬件系统和操作系统中解放出来，可以完全不去理解这些系统的结构和指令系统，只要按规定的格式书写应用程序就可以在任何支持该语言的硬件平台上执行。OpenGL 的前身是 SGI 的 IRIX GL，由于 OpenGL 的高度可重用性，已经有几十家大公司表示接受 OpenGL 作为标准图形软件接口。目前加入 OpenGL ARB (OpenGL Architecture Review Borad，即 OpenGL 体系结构审查委员会)的成员有 SGI 公司、Microsoft 公司、Intel 公司、IBM 公司、SUN 公司、DEC 公司(已由 Compaq 公司兼并)、HP 公司、AT&T 公司的 UNIX 软件实验室等。在 OpenGL ARB 的努力下，OpenGL 已经成为高性能图形和交互式视景处理的工业标准，能够在 Windows 95/98、Windows NT、Macos、Beos、OS/2 及 Unix 上应用。

作为图形硬件的软件接口，OpenGL 由几百个指令或函数组成。对程序员而言，OpenGL 是一些指令或函数的集合。这些指令允许用户对二维几何对象或三维几何对象进行说明，允许用户对对象实施操作以便把这些对象着色(Render)到帧存(Frame buffer)上。OpenGL 的大部分指令提供立即接口操作方式以便使说明的对象能够马上被画到帧存上。一个使用 OpenGL 的典型描绘程序首先在帧存中定义一个窗口，然后在此窗口中进行各种操作。在所有的指令中，有些调用用于画简单的几何对象，另外一些调用将影响这些几何对象的描绘，包括如何光照、如何着色以及如何从用户的二维或三维模型空间映射到二维屏幕。

对于 OpenGL 的实现者而言，OpenGL 是影响图形硬件操作的指令集合。如果硬件仅仅包括一个可以寻址的帧存，那么 OpenGL 就不得不几乎完全在 CPU 上实现对象的描绘，图形硬件可以包括不同级别的图形加速器，从能够画二维的直线到多边形的网栅系统到包含能够转换和计算几何数据的浮点处理器。OpenGL 可以保持数量较大的状态信息。这些状态信息可以用来指示 OpenGL 如何往帧存中画物体，有一些状态用户可以直接使用，通过调用即可获得状态值；而另外一些状态只能根据它作用在所画物体上产生的影响才可见。

OpenGL 是网络透明的，可以通过网络发送图形信息至远程机，也可以发送图形信息至多个显示屏幕，或者与其他系统共享处理任务。

OpenGL 与 DirectX、Glide、Heidi 一样是一个 3D 的 API。能否支持 OpenGL 成为检测高档图形加速卡的一个重要指标之一。

OpenGL 最早在 Windows NT 上获得支持，随着 Windows 98 的推出，将意味着 Windows 98 及 Windows 95 的所有用户都可以使用 OpenGL。事实上，Microsoft 将 OpenGL 程序库与

Windows 95、Windows 98 及 Windows NT 操作系统封装在一起，免费赠送用户。

经 OpenGL 1.0 及 OpenGL 1.1 之后，OpenGL 1.2 已经面市。事实上，OpenGL 是一个优秀的专业化 3D 的 API 的。OpenGL 已经发展成为因不同应用目的而经二次开发后的多种版本，且因不同的公司而不同。基于 OpenGL 核心函数和面向对象的编程技术 TGS 公司开发出了可以运行于 Windows NT 的 OpenGL Inventor 产品。OpenGL Inventor 提供了预建的对象和可交互的内置事件模型，可创建高级的三维场景、转换不同格式的数据文件及打印信息。SUN 公司近日发布了面向 Solaris™的新版 OpenGL 图形基础库。OpenGL 适用于下一代医学成像、地理信息、石油勘探、气候模型模拟及娱乐动画等应用范围。新版 OpenGL 提供增强的绘图性能和特性，以及运行主流图形应用所必须的可靠性和性能。

随着新版本的发行，独立软件供应商和解决方案供应商将可以利用 OpenGL 在相同应用范围内执行图形图像操作。专门为支持 3DNow 技术而设计的 OpenGL(OpenGL 1.2 支持部分 3DNow 标准)已成为高档三维工作站业内标准，越来越受到 PC 电脑游戏开发商的欢迎。

2.2 OpenGL 的工作结构

OpenGL 指令的解释模型是客户/服务器模式，即，客户(试图用 OpenGL 进行绘制工作的应用程序)向服务器(OpenGL 内核)发布命令，这些 OpenGL 命令则是由服务器来解释的。在大多数情况下，客户和服务器是运行在同一台计算机的。基于客户/服务器模式，在网络环境中则很容易使用 OpenGL，且在不同计算机上的多个客户可以得到在其他计算机上服务器的服务。这样 OpenGL 就具有网络透明性。

OpenGL的库函数被封装在 Opengl32.dll 动态链接库中。从客户应用程序发布的对 OpenGL 函数的调用首先被 Opengl32.dll 处理，在传给服务器后，被 Winsrv.dll 进一步进行处理，然后传递给 DDI(Device Driver Interface)，最后传递给视频显示驱动程序。图 2-1 显示了这一过程。

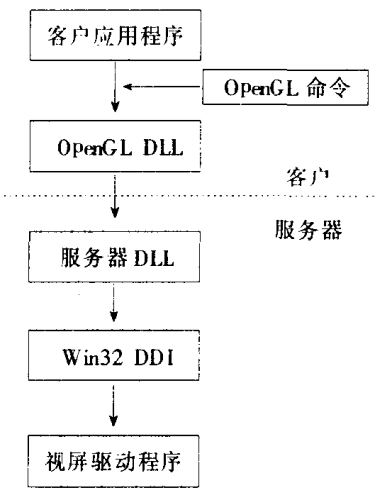

图 2-1　OpenGL 在 Windows 环境中的工作过程

2.3 OpenGL 能做什么

OpenGL能够生成真彩的三维场景，从绘制任何简单的 3D 物体到交互的动态场景，

OpenGL 都能帮助用户高效地完成这些绘制工作。具体地说，OpenGL 的功能如下：

(1) 绘制三维物体

OpenGL 以顶点为图元，由点构成线，由线及其拓扑结构构成多边形。应用点、线、多边形等基本几何图形可以绘制出任何用户想要绘制的三维形体。

(2) 观看物体

一旦完成了对场景中物体的绘制，就可以告诉 OpenGL 你想怎样观看这些物体。事实上，OpenGL 是通过一系列的变换来响应你的要求的。通过视觉变换，可以建立自己的视点(Viewport)，指定观察方向、角度及观察范围的大小。不仅如此，还可以选择自己的观察方式。通过模型变换，可以观看场景中物体的各个侧面，可进行放大、缩小、剪切等。

(3) 指定颜色模式

OpenGL 应用专门的函数和结构来指定颜色模式。OpenGL 有两种可选模式：RGBA 模式和颜色索引模式。在 RGBA 模式中，颜色值由红色、绿色、蓝色值来描述。在颜色索引模式中，颜色值则由颜色索引表中的索引值来指定。对于两种颜色模式，OpenGL 还可以选择平面着色处理或平滑着色处理。

(4) 光照

用 OpenGL 绘制的物体可以加上灯光，这使得绘制的物体与真实世界的物体极为相似。OpenGL 可以提供 4 种光，即辐射光、环境光、镜面光、漫反射光。可以指定光的颜色、光源位置等相关参数。物体被光照成类似现实物体，还与物体本身的材质有关。OpenGL 可以对物体的材质进行定义，说明它们对光的反应特性。

(5) 增强图像效果

OpenGL 提供了一系列的增强图像效果的函数，它们通过反走样、融合、雾化来增强图像效果。反走样改善图像中线的锯齿；融合可以提供半透明效果；雾化则可以模糊场景，使场景更逼真。可以对整个场景进行反走样处理，也可以实现类似照相技术中的对焦处理，还可以实现运动模糊等特殊效果。

(6) 管理位图和图像

OpenGL 可以管理两种类型的位图图像。其一是单色的位图，主要用于正确地生成字符等简单的图像；其二是真彩位图，它们可以按各种方式在屏幕和内存间进行传递。

(7) 纹理映射

通过众多的彩色多边形创建的物体往往因为表现其细节不够而显得不够真实。基于此，OpenGL 可以让程序员应用纹理映射(Texture mapping，以点映射来包裹一个物体)把真实图像映射到物体的表面。

(8) 制作动画

为了生成平滑的动画，动态场景需要在它们被传递到屏幕之前在内存中建立帧，OpenGL 采用双缓存技术。这种技术是先在内存创建好两帧基本场景，当一帧在屏幕上显示时，在内存中同时对即将要在屏幕上显示的后一帧(第二帧)进行修改。当第二帧完成之后，就清除缓存中的先前帧(第一帧)以便显示后一帧，同时，第二帧占用的缓存也被清刷。这样循环交替下去，以致产生平滑动画。

(9) 交互技术

交互技术是 OpenGL 的一个重要应用。OpenGL 提供 3 种工作模式：绘图模式、选择模式和反馈模式。绘图模式完成场景的绘制，可以借助于物体的几何参数及运动控制参数、场景的

观察参数、光照参数与材质参数、纹理参数、OpenGL 函数的众多常量控制参数、时间参数等和 Windows 对话框、菜单、外部设备等构成时实交互的程序系统。在选择模式下，则可以对物体进行命名，选择命名的物体，控制对命名物体的绘制。而反馈模式则给程序设计提供了程序运行的信息，这些信息也可反馈给用户，告诉用户程序的运行状况和监视程序的运行进程。

2.4 OpenGL 是怎样工作的

作为图形硬件的软件接口，OpenGL 最主要的工作就是将二维及三维物体描绘至帧缓存。这些物体由一系列的描述物体几何性质的顶点(Vertex)或描述图像的像素(Pixel)组成。OpenGL 执行一系列的操作把这些数据最终转化成像素数据并在帧缓存中形成最后的结果。下面对 OpenGL 的工作作概要性介绍。

(1) 图元操作与指令

OpenGL 能够从多种可选择的模式画图元(Primitive)，而且一种模式的设置一般不会影响其他模式的设置。无论发生什么情况，指令总是被顺序处理的，也就是说，一个图元必须完全画完之后，后继图元才能影响帧存。

图元由一组顶点定义。该组顶点既可以只包含一个顶点，也可以包含多个顶点。顶点的说明由位置坐标、颜色值、法向量和纹理坐标组成。每个顶点可以被顺序或以相同的方式独立地处理。每个顶点根据其为二维或三维顶点可以分别使用 2 个坐标、3 个坐标或 4 个坐标。此外，当前法线、当前纹理坐标以及当前颜色值可以在处理每个顶点的过程中被使用或改变。当前法线是一个三维向量，被用于光照计算。纹理坐标决定如何把纹理图像映射到图元。颜色与每个顶点有关，相关的颜色或者是当前颜色或者是由光照产生的颜色，这取决于是否允许光照。类似地，纹理坐标也与每个顶点坐标有关。

在 OpenGL 中，几何对象是根据 glBegin()/glEnd()函数对之间所包含的一系列指定顶点的位置坐标、颜色值、法向量值和纹理坐标画出的。这样的几何对象有点线段、循环线段、分离线段、多边形、三角形、三角形扇、分离的三角形、四边形及分离的四边形。

(2) 图形控制

OpenGL 提供诸如变换矩阵、光照、反走样方法、像素操作等参数来控制二维和三维图形的绘制。然而，它并不提供一个描述或建立复杂几何物体的手段。OpenGL 提供的是怎样画复杂物体的机制而非描绘复杂物体本身的面面俱到的工具。

(3) OpenGL 指令解释模型

OpenGL 的指令解释模型是客户/服务器模式。

(4) OpenGL 基本操作

OpenGL 可以在具有不同图形能力和性能的图形工作站平台及微机上运行。图 2-2 给出了 OpenGL 的绘制原理图。

用户指令从左侧进入 OpenGL。指令分为两部分。一部分画指定的几何物体，另一部分则指示在不同的阶段怎样处理几何物体。许多指令很可能被排列在显示列表(Display list)中，在后续时间里对其进行处理。通过评价器(Evaluator)计算输入值的多项式函数来为近似曲线和曲面等几何物体提供有效手段，然后由顶点描述的几何图元进行操作。在此阶段，对顶点进行转换、光照，并把图元剪切到观察体(Frustum)中，为下一步光栅化(Rasterization)做准备。光栅化产生一系列图像的帧缓存地址和图元的二维描述值，其生成结果称为基片(Fragment)，

每个基片适合于在最后改变帧存之前对单个的基片进行操作。这些操作包括根据先前储存的深度值进行有条件地更新帧缓存,进行各种测试以及融合即将处理的基片颜色与已经储存的颜色,进行屏蔽,对基片进行逻辑操作和淡化(Dithering)。

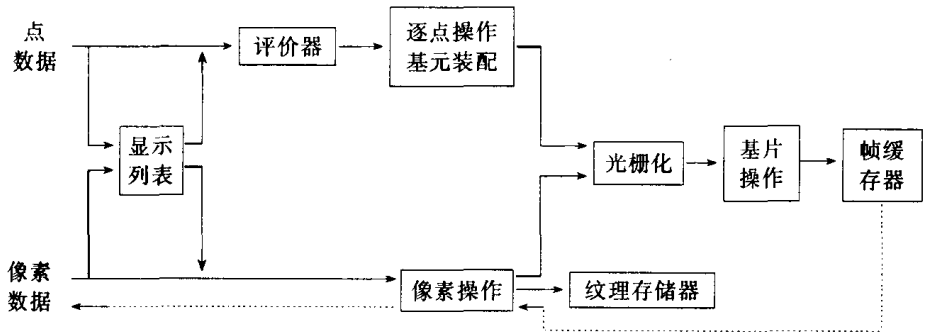

图 2-2 OpenGL 的绘制原理

图像像素数据的处理包括像素、位图、影像等,它们经过像素操作之后直接进入光栅化阶段。

由于 OpenGL 的几何图元是由顶点描述的,这样可以便于逐点操作并按其数据装配成基元,然后经光栅化形成基片。对于像素数据,其结果还可以存储在纹理用的内存中,然后从纹理内存中取出像素信息进行光栅化。

OpenGL 显示列表事实上是一组函数,它们被存储以使 OpenGL 在后继时间内能够处理。

OpenGL 评价器也是许多特殊的函数。这些函数允许你采用一个或两个变量的多项式映射来产生顶点坐标、法线坐标、纹理坐标及颜色,生成结果传递给执行管道(Pipeline)。评价器接口提供在 OpenGL 之上建立更一般的曲线和曲面包的基础。在 OpenGL 中提供评价器而不是更复杂的 NURBS 接口的优点在于,在表示非 NURBS 曲线和曲面以及利用特殊的表面性质时,仍然能对多项式评价器进行有效地利用,而不必将其转换为 NURBS 表示。

光栅化包含几何和物理映射两部分。几何操作是将图元转化成二维图像;物理操作是计算图像每个点的颜色和深度等信息。因而,光栅化一个图元由两部分操作组成。第一部分是决定窗口坐标(Window coordinates)中一个整数栅格的哪些方块由图元占有;第二部分是为每个这样的方块计算它们的颜色值和深度值。计算的结果将被传递到 OpenGL 的下一过程,并用此信息更新帧存中的适当单元。在 OpenGL 中,栅格方块不一定是方形的,光栅化的规则不受实际的栅格方块的宽高比的限制。当然,非方形栅格的显示必然会引起光栅化的点和线段在一个方向比在另一个方向显得宽些。方形的基片可以简化反混淆和纹理的难度。

2.5 OpenGL 图形操作步骤

运用 OpenGL 进行主要的图形操作以及最终在计算机屏幕上绘制出三维场景的基本步骤是:

第一,设置像素格式。像素格式告诉 OpenGL 绘制风格、颜色模式、颜色位数、深度位数等等重要信息。

第二,建立模型。根据基本图元建立景物的三维模型,并对模型进行数学描述。

第三,舞台布置。把景物放置在三维空间的适当位置,设置三维透视视觉体以观察场景。

第四,效果处理。设置物体的材质(颜色、光学性能及纹理映射方式等),加入光照及光照条件。

第五,光栅化。把景物及其颜色信息转化为可在计算机屏幕上显示的像素信息。

2.6 OpenGL 函数及结构

不同的公司基于自己的用途所开发的 OpenGL 应用函数不尽相同。下面以 Microsoft 的 Windows 95(98)及 Windows NT 为支撑平台,说明 OpenGL 函数。

(1) OpenGL 核心函数

OpenGL 核心函数有 100 多个,每个函数以 gl 开头。这些函数是最基本的,可以运行于任何的 OpenGL 工作平台。可用这些函数创建二维和三维几何形体,设置视点,建立视觉体,设置颜色及材质,建立灯光,进行纹理映射,反走样,处理融合,雾化场景等。它们可以接受不同的参数因而可派生出 300 多个函数。

(2) OpenGL 实用库函数

OpenGL 实用库函数以 glu 开头。这些函数事实上是基于 OpenGL 核心函数,提供对辅助函数特性的支持,并且执行了核心的 OpenGL 交互,因而是比核心函数更高一层的函数,也更具有通用性。实用函数完成初始化,管理纹理映像,变换坐标,细化多边形,应用回调函数,细化物体,申明回调及细化,着色简单曲面(椭球、圆柱等),产生曲线、曲面及处理错误。像 OpenGL 核心函数一样,它们可以运行于任何 OpenGL 平台。

(3) 辅助库函数

辅助库函数有 31 个,这些函数以 aux 开头。它们是一类特殊的 OpenGL 函数,是帮助初学者尽快进入 OpenGL 编程而做简单练习之用。因此,并不能在所有的平台上使用,但 Windows 95(98)及 Windows NT 支持它们。这些函数简化了像素格式的设置,集成了窗口管理系统的生成、控制及背景过程,简化了可定制的输入事件及交互手段,可绘制简单的典型的三维物体等。初学者采用它们仅仅只关心怎样去进行几何物体的数学描述,且不需要 Windows 编程的基本知识。在产品程序中尽量不要用它们,主要有以下几个原因:第一,消息循环不是在代码中进行;第二,没有为其他所需的消息添加句柄的方法;第三,几乎不支持调色盘。

本书是为 Windows 编程人员写的,不仅仅是对 OpenGL 作入门性的介绍,因此本书不准备讲述辅助库函数的用法,在大量的程序中也不准备采用它们。

(4) Windows 专用函数

这些函数以 wgl 开头,是用于连接 OpenGL 和 Windows 窗口系统的。用它们可以管理着色描述表(Rende context)及显示列表,扩展功能,管理字体位图等。这些函数如下:

wglCopyContext	wglCreateContext
wglCreateLayerContext	wglDeleteContext
wglDescribeLayerPlane	wglGetCurrentContext
wglGetCurrentDC	wglGetLayerPaletteEntries
wglGetProcAddress	wglMakeCurrent
wglRealizeLayerPalette	wglSetLayerPaletteEntries
wglShareLists	wglSwapLayerBuffers
wglUseFontBitmaps	wglUseFontOutlines

(5) Win32 API 函数

共 6 个 Win32 API 函数,用于处理像素格式及缓冲。这些函数如下:

ChoosePixelFormat

DescribePixelFormat

GetEnhMetaFilePixelFormat

GetPixelFormat

SetPixelFormat

SwapBuffers

(6) OpenGL 结构

OpenGL 结构共 4 个,现列举如下:

GLYPHMETRICSFLOAT

LAYERPLANEDESCRIPTOR

PIXELFORMATDESCRIPTOR

POINTFLOAT

必须指出的是,OpenGL 的每个函数几乎都有多个状态变量可供选择,从而可利用它们编制出非凡的三维图像。本书不是一本参考手册,而是一本编程指南,即介绍如何把如此众多的分散的功能及状态变量有机地结合在一起,使它们相互联系并协同工作,指导读者完成图像制作。另外,本书也不准备对每个函数一一进行细致的解释,而是把这项工作留给读者去查阅 Windows 在线帮助。

2.7 OpenGL 数据类型

有过 Windows 编程经历的人都知道,为了使程序员方便地处理数值,Windows 定义了多种特别的数据类型,如 HDC、HBRUSH、COLORREF、RECT、POINT、UINT 等等。OpenGL 也定义了许多编程人员可应用的数据类型,现列举如下表。

表 2-1

数据类型	等价语言数据类型
Glbyte	sign char
Glshort	short
Glint	long
Glsizei	long
Glfloat	float
Glclampf	float
Gldouble	double
Glclampd	double
Glubyte	unsigned char
Glboolean	unsigned char
Glushort	unsigned short
Gluint	unsigned long
Glenum	unsigned long
Glbitfield	unsigned long
Glvoid	void
HGLRC	HGDIOBJ

2.8 OpenGL 句法

前面已经指出，OpenGL 的核心函数只有 100 多个，它们又是怎样被扩展成 300 多个函数的呢？这些函数不同的形式主要来自于函数不同的后缀，而后缀又是由函数参数所采用的数据类型决定的。比如 glVertex2f()函数(用来描述一个二维的以单精度浮点为参数的一个顶点)，2f 则是后缀。该后缀同时指示了用户只能用两个单精度浮点数作为其参数。glVertex 命令实际上有 24 种不同的形式。现列举如下：

```
glVertex2d( GLdouble x, GLdouble y );
glVertex2f( GLfloat x, GLfloat y );
glVertex2i( GLint x, GLint y );
glVertex2s( GLshort x, GLshort y );
glVertex3d( GLdouble x, GLdouble y, GLdouble z );
glVertex3f( GLfloat x, GLfloat y, GLfloat z );
glVertex3i( GLint x, GLint y, GLint z );
glVertex3s( GLshort x, GLshort y, GLshort z );
glVertex4d( GLdouble x, GLdouble y, GLdouble z, GLdouble w );
glVertex4f( GLfloat x, GLfloat y, GLfloat z, GLfloat w );
glVertex 4i( GLint x, GLint y, GLint z, GLint w );
glVertex4s( GLshort x, GLshort y, GLshort z, GLshort w );
glVertex2dv( const GLdouble *v );
glVertex2fv( const GLfloat *v );
glVertex2iv( const GLint *v );
glVertex2sv( const GLshort *v );
glVertex3dv( const GLdouble *v );
glVertex3fv( const GLfloat *v );
glVertex3iv( const GLint *v );
glVertex3sv( const GLshort *v );
glVertex4dv( const GLdouble *v );
glVertex4fv( const GLfloat *v );
glVertex4iv( const GLint *v );
glVertex4fv( const GLshort *v );
```

可见，不同的形式仅仅是后缀不同而已。表 2-2 列出了各函数名后缀及其含义。

表 2-2

后缀	函数采用的参数数据类型
b	GLbyte
d	GLdouble 或 GLclampd
f	GLfloat 或 GLclampf
i	GLint 或 Glsizei

（续表）

s	GLshort
ub	GLubyte 或 Glboolean
ui	GLuint 或 GLenum 或 GLbitfield
us	GLushort
bv	向量，类型为 GLbyte
dv	向量，类型为 GLdouble 或 GLclampd
fv	向量，类型为 GLfloat 或 GLclampf
iv	向量，类型为 GLint 或 Glsizei
sv	向量，类型为 GLshort
ubv	向量，类型为 GLubyte 或 Glboolean
uiv	向量，类型为 GLuint 或 GLenum 或 GLbitfield
usv	向量，类型为 GLushort

2.9 OpenGL 状态变量

在 OpenGL 中，除核心库函数、实用库函数和辅助库函数外，OpenGL 本身又提供了许多状态变量，这些状态变量均作为 OpenGL 函数的参数，OpenGL 将根据具体的参数值决定如何绘制图形、如何显示图形和如何处理错误。状态变量事实上就是 OpenGL 常量，以大写的 GL 开头。

不仅可以将 OpenGL 状态变量作为参数赋给函数，还可以对其进行查询和获取。一般地，当设置一个 OpenGL 状态之后，如果不重新设置与之相应的状态值，那么所设置的状态值将一直有效。为了提供更高的灵活性，OpenGL 可以对状态变量进行保存和恢复。

下面的 4 个函数可以用来获取布尔型、整型、单精度类型和双精度类型的状态变量：

```
void glGetBooleanv( GLenum pname, GLboolean * params );
void glGetDoublev( GLenum pname, GLdouble * params );
void glGetFloatv( GLenum pname, GLfloat * params );
void glGetIntegerv( GLenum pname, GLint * params );
```

比如，要获取当前设备描述表所支持的直线的宽度，可以采用如下语句来完成：

```
GLfloat lw[2];
glGetFloatv(GL_LINE_WIDTH_RANGE,lw);
```

其中，lw[0]表示最小宽度值，lw[1]表示最大宽度值。又如，要获取设备描述表所支持的点的大小，可用如下语句：

```
GLfloat fPointSize[2];
glGetFloatv(GL_POINT_SIZE_RANGE,fPointSize);
```

可见，参数 pname 表示要获取的值，而参数 params（存储地址）则接受返回值。参数 pname 可以取值如下：

累积缓存操作：

GL_ACCUM_ALPHA_BITS　　GL_ACCUM_BLUE_BITS
GL_ACCUM_CLEAR_VALUE　　GL_ACCUM_GREEN_BITS
GL_ACCUM_RED_BITS

颜色与 Alpha 测试操作：

GL_RED_BIAS　　GL_RED_BITS
GL_RED_SCALE　　GL_GREEN_BIAS
GL_GREEN_BITS　　GL_GREEN_SCALE
GL_BLUE_BIAS　　GL_BLUE_BITS
GL_BLUE_SCALE　　GL_CURRENT_COLOR
GL_CURRENT_INDEX　　GL_ALPHA_BIAS
GL_ALPHA_BITS　　GL_ALPHA_SCALE
GL_ALPHA_TEST　　GL_ALPHA_TEST_FUNC
GL_ALPHA_TEST_REF

属性堆栈：

GL_ATTRIB_STACK_DEPTH　　GL_MAX_ATTRIB_STACK_DEPTH

法　线：

GL_AUTO_NORMAL　　GL_CURRENT_NORMAL
GL_NORMALIZE

辅助缓存：

GL_AUX_BUFFERS

融合操作：

GL_BLEND　　GL_BLEND_DST
GL_BLEND_SRC

剪切操作：

GL_CLIP_PLANEi　　GL_MAX_CLIP_PLANES
GL_SCISSOR_BOX　　GL_SCISSOR_TEST
GL_CULL_FACE　　GL_CULL_FACE_MODE
GL_MAX_CLIP_PLANES

颜色缓存：

GL_RGBA_MODE　　GL_COLOR_CLEAR_VALUE
GL_COLOR_WRITEMASK　　GL_INDEX_BITS

GL_INDEX_CLEAR_VALUE　　GL_INDEX_MODE
GL_INDEX_OFFSET　　GL_INDEX_SHIFT
GL_INDEX_WRITEMASK

颜色与材质：
GL_COLOR_MATERIAL　　GL_COLOR_MATERIAL_FACE
GL_COLOR_MATERIAL_PARAMETER

当前光栅属性：
GL_CURRENT_RASTER_COLOR　　GL_CURRENT_RASTER_DISTANCE
GL_CURRENT_RASTER_INDEX　　GL_CURRENT_RASTER_POSITION
GL_CURRENT_RASTER_TEXTURE_COORDS
GL_CURRENT_RASTER_POSITION_VALID
GL_CURRENT_TEXTURE_COORDS

深度缓存：
GL_DEPTH_BIAS　　GL_DEPTH_BITS
GL_DEPTH_CLEAR_VALUE　　GL_DEPTH_FUNC
GL_DEPTH_RANGE　　GL_DEPTH_SCALE
GL_DEPTH_TEST　　GL_DEPTH_WRITEMASK

淡　化：
GL_DITHER

帧　存：
GL_DOUBLEBUFFER　　GL_DRAW_BUFFER
GL_READ_BUFFER

点、直线与多边形：
GL_EDGE_FLAG　　GL_LINE_SMOOTH
GL_LINE_SMOOTH_HINT　　GL_LINE_STIPPLE
GL_LINE_STIPPLE_PATTERN　　GL_LINE_STIPPLE_REPEAT
GL_LINE_WIDTH　　GL_LINE_WIDTH_GRANULARITY
GL_LINE_WIDTH_RANGE　　GL_POINT_SIZE
GL_POINT_SIZE_GRANULARITY　　GL_POINT_SIZE_RANGE
GL_POINT_SMOOTH　　GL_POINT_SMOOTH_HINT
GL_POLYGON_MODE　　GL_POLYGON_SMOOTH
GL_POLYGON_SMOOTH_HINT　　GL_POLYGON_STIPPLE
GL_FRONT_FACE

雾：

GL_FOG　　GL_FOG_COLOR
GL_FOG_DENSITY　　GL_FOG_END
GL_FOG_HINT　　GL_FOG_INDEX
GL_FOG_MODE　　GL_FOG_START

光　照：

GL_LIGHTi　　GL_LIGHTING
GL_LIGHT_MODEL_AMBIENT　　GL_LIGHT_MODEL_LOCAL_VIEWER
GL_LIGHT_MODEL_TWO_SIDE　　GL_MAX_LIGHTS

显示列表属性：

GL_LIST_BASE　　GL_LIST_INDEX
GL_LIST_MODE　　GL_MAX_LIST_NESTING

逻辑操作：

GL_LOGIC_OP　　GL_LOGIC_OP_MODE

评价器：

GL_MAP1_COLOR_4　　GL_MAP1_GRID_DOMAIN
GL_MAP1_GRID_SEGMENTS　　GL_MAP1_INDEX
GL_MAP1_NORMAL　　GL_MAP1_TEXTURE_COORD_1
GL_MAP1_TEXTURE_COORD_2　　GL_MAP1_TEXTURE_COORD_3
GL_MAP1_TEXTURE_COORD_4　　GL_MAP1_VERTEX_3
GL_MAP1_VERTEX_4　　GL_MAP2_COLOR_4
GL_MAP2_GRID_DOMAIN　　GL_MAP2_GRID_SEGMENTS
GL_MAP2_INDEX　　GL_MAP2_NORMAL
GL_MAP2_TEXTURE_COORD_1　　GL_MAP2_TEXTURE_COORD_2
GL_MAP2_TEXTURE_COORD_3　　GL_MAP2_TEXTURE_COORD_4
GL_MAP2_VERTEX_3　　GL_MAP2_VERTEX_4
GL_MAX_EVAL_ORDER

模板缓存：

GL_MAP_STENCIL　　GL_STENCIL_BITS
GL_STENCIL_CLEAR_VALUE　　GL_STENCIL_FAIL
GL_STENCIL_FUNC　　GL_STENCIL_PASS_DEPTH_FAIL
GL_STENCIL_PASS_DEPTH_PASS　　GL_STENCIL_REF
GL_STENCIL_TEST　　GL_STENCIL_VALUE_MASK
GL_STENCIL_WRITEMASK

矩阵操作：

GL_MATRIX_MODE

GL_MAX_MODELVIEW_STACK_DEPTH

GL_MAX_PROJECTION_STACK_DEPTH

GL_MAX_VIEWPORT_DIMS

GL_MODELVIEW_MATRIX

GL_MODELVIEW_STACK_DEPTH

GL_PERSPECTIVE_CORRECTION_HINT

GL_PROJECTION_MATRIX

GL_PROJECTION_STACK_DEPTH

GL_VIEWPORT

名称堆栈：

GL_MAX_NAME_STACK_DEPTH

GL_NAME_STACK_DEPTH

像素操作：

GL_MAX_PIXEL_MAP_TABLE

GL_MAP_COLOR

GL_PACK_ALIGNMENT

GL_PACK_LSB_FIRST

GL_PACK_ROW_LENGTH

GL_PACK_SKIP_PIXELS

GL_PACK_SKIP_ROWS

GL_PACK_SWAP_BYTES

GL_PIXEL_MAP_A_TO_A_SIZE

GL_PIXEL_MAP_B_TO_B_SIZE

GL_PIXEL_MAP_G_TO_G_SIZE

GL_PIXEL_MAP_I_TO_A_SIZE

GL_PIXEL_MAP_I_TO_B_SIZE

GL_PIXEL_MAP_I_TO_G_SIZE

GL_PIXEL_MAP_I_TO_I_SIZE

GL_PIXEL_MAP_I_TO_R_SIZE

GL_PIXEL_MAP_R_TO_R_SIZE

GL_PIXEL_MAP_S_TO_S_SIZE

GL_SUBPIXEL_BITS

GL_UNPACK_ALIGNMENT

GL_UNPACK_LSB_FIRST

GL_UNPACK_ROW_LENGTH

GL_UNPACK_SKIP_PIXELS

GL_UNPACK_SKIP_ROWS

GL_UNPACK_SWAP_BYTES

GL_ZOOM_X

GL_ZOOM_Y

纹理操作：

GL_MAX_TEXTURE_STACK_DEPTH

GL_TEXTURE_1D

GL_TEXTURE_2D

GL_TEXTURE_ENV_COLOR

GL_TEXTURE_ENV_MODE

GL_TEXTURE_GEN_S

GL_TEXTURE_GEN_T

GL_TEXTURE_GEN_R

GL_TEXTURE_GEN_Q

GL_TEXTURE_MATRIX

GL_TEXTURE_STACK_DEPTH

GL_MAX_TEXTURE_SIZE

着色模式：

GL_RENDER_MODE

GL_SHADE_MODEL

各状态变量的含义可以通过在线帮助来获得。但应该指出，要完全弄清它们的含义并灵活地应用它们并不是一两天的事，因此请读者暂时可以不去理会它们。随着对 OpenGL 的熟

悉和应用的需要,将会在应用中逐步认识它们。

OpenGL 还可以返回一些特殊的状态,这些返回状态本身并不仅是基本数据类型:

```
void glGetClipPlane( GLenum plane, GLdouble * equation );
GLenum glGetError( void );
void glGetLightfv( GLenum light, GLenum pname, GLfloat * params );
void glGetLightiv( GLenum light, GLenum pname, GLint * params );
void glGetMapdv( GLenum target, GLenum query, GLdouble * v );
void glGetMapfv( GLenum target, GLenum query, GLfloat * v );
void glGetMapiv( GLenum target, GLenum query, GLint * v );
void glGetMaterialfv( GLenum face, GLenum pname, GLfloat * params );
void glGetMaterialiv( GLenum face, GLenum pname, GLint * params );
void glGetPixelMapfv( GLenum map, GLfloat * values );
void glGetPixelMapuiv( GLenum map, GLuint * values );
void glGetPixelMapusv( GLenum map, GLushort * values );
void glGetPolygonStipple( GLubyte * mask );
const GLubyte * glGetString( GLenum name );
void glGetTexEnvfv( GLenum target, GLenum pname, GLfloat * params );
void glGetTexEnviv( GLenum target, GLenum pname, GLint * params );
void glGetTexGendv( GLenum coord, GLenum pname, GLdouble * params );
void glGetTexGenfv( GLenum coord, GLenum pname, GLfloat * params );
void glGetTexGeniv( GLenum coord, GLenum pname, GLint * params );
void glGetTexImage( GLenum target, GLint level, GLenum format,
                    GLenum type, GLvoid * pixels );
void glGetTexLevelParameterfv( GLenum target, GLint level, GLenum pname, GLfloat * params );
void glGetTexLevelParameteriv( GLenum target, GLint level, GLenum pname, GLint * params );
void glGetTexParameterfv( GLenum target, GLenum pname, GLfloat * params );
void glGetTexParameteriv( GLenum target, GLenum pname, GLint * params );
```

OpenGL 可以查询变量是否被激活:

```
GLboolean glIsEnabled( GLenum cap );
```

如果 cap 参数被激活则返回 GL_TRUE。cap 参数主要表示一些有关 OpenGL 的功能状态,它们是:

GL_ALPHA_TEST
GL_AUTO_NORMAL
GL_BLEND
GL_CLIP_PLANEi
GL_COLOR_MATERIAL
GL_CULL_FACE
GL_DEPTH_TEST
GL_DITHER
GL_FOG
GL_LIGHTi
GL_LIGHTING
GL_LINE_SMOOTH
GL_LINE_STIPPLE
GL_LOGIC_OP
GL_MAP1_COLOR_4
GL_MAP1_INDEX
GL_MAP1_NORMAL
GL_MAP1_TEXTURE_COORD_1

GL_MAP1_TEXTURE_COORD_2　　GL_MAP1_TEXTURE_COORD_3
GL_MAP1_TEXTURE_COORD_4　　GL_MAP1_VERTEX_3
GL_MAP1_VERTEX_4　　GL_MAP2_COLOR_4
GL_MAP2_INDEX　　GL_MAP2_NORMAL
GL_MAP2_TEXTURE_COORD_1　　GL_MAP2_TEXTURE_COORD_2
GL_MAP2_TEXTURE_COORD_3　　GL_MAP2_TEXTURE_COORD_4
GL_MAP2_VERTEX_3　　GL_MAP2_VERTEX_4
GL_NORMALIZE　　GL_POINT_SMOOTH
GL_POLYGON_SMOOTH　　GL_POLYGON_STIPPLE
GL_SCISSOR_TEST　　GL_STENCIL_TEST
GL_TEXTURE_1D　　GL_TEXTURE_2D
GL_TEXTURE_GEN_Q　　GL_TEXTURE_GEN_R
GL_TEXTURE_GEN_S　　GL_TEXTURE_GEN_T

当需要这些功能时，就可以用 glEnable()函数来激活，比如，要激活第零号灯：

```
glEnable(GL_LIGHT0);
```

如果要取消一项功能，则应调用 glDisable()函数，比如，取消第零号灯：

```
glDisable(GL_LIGHT0);
```

这些特定功能是 OpenGL 制作的基础。

OpenGL 可以保存和恢复状态变量，用如下两条语句来实现：

```
void glPushAttrib( GLbitfield mask );
void glPopAttrib( void );
```

mask 参数可以取如下常量：

GL_ACCUM_BUFFER_BIT　　GL_COLOR_BUFFER_BIT
GL_CURRENT_BIT　　GL_DEPTH_BUFFER_BIT
GL_ENABLE_BIT　　GL_EVAL_BIT
GL_FOG_BIT　　GL_HINT_BIT
GL_LIGHTING_BIT　　GL_LINE_BIT
GL_LIST_BIT　　GL_PIXEL_MODE_BIT
GL_POINT_BIT　　GL_POLYGON_BIT
GL_POLYGON_STIPPLE_BIT　　GL_SCISSOR_BIT
GL_STENCIL_BUFFER_BIT　　GL_TEXTURE_BIT
GL_TRANSFORM_BIT　　GL_VIEWPORT_BIT

比如，要保存和恢复当前光照与否的属性，使得在绘制一些平面图时不受光照的影响，可以按如下方式进行：

```
//    保存当前光照与否的设置情况
    glPushAttrib(GL_LIGHTING_BIT);
    //    无论怎样，均取消光照，DrawMyObjects()所绘制的物体将没有光照
```

```
    glDisable(GL_LIGHTING);
    //    绘制物体,通常为平面或直线或点
    DrawMyObjects();
glPopAttrib();
//    恢复光照设置
```

2.10 齐次坐标表示

glVertex()函数有24种不同的形式。在三维空间中,一个点是用一个三维向量(x,y,z)来表示的,为什么用4个分量来表示一个点呢? 如合法命令glVetex4f(1.0f,2.0f,1.0f,4.0f)。为了便于统一处理,在OpenGL中,三维空间的点都用齐次坐标(Homogeneous Coordinate)表示。齐次坐标的一般形式为(x,y,z,w)。三维空间点的齐次坐标表示法则如下:

(1) 二维空间的点(x,y)的齐次坐标表示为(x,y,0.0,1.0);

(2) 三维空间的点(x,y,z)的齐次坐标表示为(x,y,z,1.0);

(3) 当w不为0时,即为三维空间的点(x/w,y/w,z/w)。当w为0时,表示一个无穷远点。

第 3 章　极小化 OpenGL

到目前为止读者对 OpenGL 已有了一个初步的认识。假定读者已安装了 Visual C++ 2.0 以上的版本，并能在 Windows 95 的系统目录下找到 Opengl32.dll 及 glu32.dll。这样读者就可以工作了。注意一切都是从零开始。本书所有程序是在 Windows 98 及 Visual C++ 5.0 上开发的，OpenGL 版本为 1.2。

在本章，将介绍怎样极小化一个 OpenGL 应用程序，并且读者将会看到为一个物体指定颜色是一件十分容易的事。

3.1　OpenGL 像素格式

从图 2-2 中已经知道，无论进入 OpenGL 的是何种数据，OpenGL 最终都将进行像素操作，即把像素写到帧存上去。OpenGL 应该知道怎样操纵这些像素，因此必须设置像素的属性。在 Microsoft Windows NT 和 Windows 95 上采用数据结构 PIXELFORMATDESCRIPTOR 来传送像素的属性。这些属性使 OpenGL 明确：像素缓存是单缓存还是双缓存；像素数据的颜色模式是 RGBA 模式或颜色索引模式；颜色数据的位数；深度缓存的位数；模板缓存的位数；交叠面类型；掩膜的可见性等。

每一个 OpenGL 显示设备都支持特定数目的像素格式。当然，这些可获得的像素格式是基于显示设备的能力的。一个指定的像素格式必须通过 PIXELFORMATDESCRIPTOR 结构得到描述。下面我们对该结构作一些说明。

```
typedef struct tagPIXELFORMATDESCRIPTOR {
        WORD nSize;
        WORD nVersion;
        DWORD dwFlags;
        BYTE iPixelType;
        BYTE cColorBits;
        BYTE cRedBits;
        BYTE cRedShift;
        BYTE cGreenBits;
        BYTE cGreenShift;
        BYTE cBlueBits;
        BYTE cBlueShift;
        BYTE cAlphaBits;
        BYTE cAlphaShift;
        BYTE cAccumBits;
        BYTE cAccumRedBits;
        BYTE cAccumGreenBits;
        BYTE cAccumBlueBits;
```

```
    BYTE cAccumAlphaBits;
    BYTE cDepthBits;
    BYTE cStencilBits;
    BYTE cAuxBuffers;
    BYTE iLayerType;
    BYTE bReserved;
    DWORD dwLayerMask;
    DWORD dwVisibleMask;
    DWORD dwDamageMask;
} PIXELFORMATDESCRIPTOR;
```

其成员的含义如下：

nSize:说明该结构的长度；

nVersion：说明该结构的版本号；

dwFlags：告诉 OpenGL 怎样处理像素，可以是下面这些常量的逻辑或。这些常量事实上是一些标识位，它们可以是：

PFD _ DRAW _ TO _ WINDOW	FD _ DRAW _ TO _ BITMAP
PFD _ SUPPORT _ GDI	PFD _ SUPPORT _ OPENGL
PFD _ GENERIC _ FORMAT	PFD _ NEED _ PALETTE
PFD _ NEED _ SYSTEM _ PALETTE	PFD _ DOUBLEBUFFER
PFD _ STEREO	PFD _ SWAP _ LAYER _ BUFFERS

iPixelType:说明颜色模式。只能采用 PFD _ TYPE _ RGBA 及 PFD _ TYPE _ COLORINDEX 中的一个。前者告诉 OpenGL 采用 RGBA 模式，后者告诉 OpenGL 采用颜色索引模式。

cColorBits:颜色位数。比如，8 位颜色可以提供 256 种颜色。

cRedBits ，cRedShift，cGreenBits，cGreenShift，cBlueBits，cBlueShift 一般不采用，均置为零。

cAlphaBits:RGBA 颜色缓存中 Alpha 位数。

cAlphaShift：现已不采用，置为零。

cAccumBits:累积缓存的位数。

cAccumRedBits，cAccumGreenBits，cAccumBlueBits，cAccumAlphaBits:不采用，置为零。

cDepthBits:深度缓存的位数。

cStencilBits:模板缓存的位数。

cAuxBuffers:辅助缓存的位数。一般被置为零。

iLayerType:说明层面类型。选择下面 3 者之一：PFD _ MAIN _ PLANE，PFD _ OVERLAY _ PLANE 及 PFD _ UNDERLAY _ PLANE。必须指出，这个成员也是为最早的 OpenGL 设计的，可以忽略，一般被置为 0。

bReserved，dwLayerMask，dwVisibleMask，dwDamageMask 必须为 0。

下面对成员 dwFlags 所用的常量进行说明。

PFD _ DRAW _ TO _ BITMAP：	应用程序把绘制结果写向位图。
PFD _ DRAW _ TO _ WINDOW：	应用程序把绘制结果输出到屏幕。
PFD _ DOUBLEBUFFER：	支持双缓存技术，应用 OpenGL 生成动画必须有

	此设置。
PFD_GENERIC_FORMAT：	选择通用像素格式，这种格式能被 OpenGL 所支持。如果 不进行此项设置，则像素格式必须获得相关驱动程序或硬件的支持。
PFD_NEED_PALETTE：	如果有此设置，则必须建立逻辑调色盘。
PFD_NEED_SYSTEM_PALETTE:	采用系统支持的一个硬件调色盘。但是在应用程序中，必须通过调用 SetSystemPaletteUse()函数来强迫系统在调色盘和逻辑调色盘之间建立一一对应关系。如果 OpenGL 硬件支持多个硬件调色盘，并且设备驱动程序能够为 OpenGL 分配多余的硬件调色盘，该设置就自动清除。
PFD_STEREO：	选择体视缓存。
PFD_SUPPORT_GDI:	支持 GDI 绘制函数。在双缓存模式下不能使用。
PFD_SUPPORT_OPENGL:	支持 OpenGL 函数。
PFD_SWAP_LAYER_BUFFERS:	在双缓存的多层面像素格式中，指示一个设备是否一次只清除一个层面。否则，所有的层面都将作为一组一并清除。同时，也只有设置此标识，wglSwapLayerBuffers()函数才会获得支持。

3.2 着色描述表

任何一个 Windows 程序都必须处理设备描述表(Device Context)，它告诉 Windows 怎样在一个窗口中显示图形信息。一个设备描述表(DC)说明了笔和画刷的颜色、绘制模式调色盘信息、映射模式以及其他的 Windows 必须知道怎样显示图形的属性。比如，要在一个窗口中显示文字，就必须在该窗口中创建一个 DC，为绘制函数提供一个指向该 DC 的句柄等等。与其他的 Windows 应用程序一样，OpenGL 应用程序也必须应用 DC。不过，我们将其称为着色描述表(Rendering Context)。着色描述表(RC)保存了与系统发生联系的重要信息。一个应用程序必须有一个着色描述表。

3.3 设置像素格式

3.3.1 填充 PIXELFORMATDESCRIPTOR 结构

设置像素格式，首先要填充 PIXELFORMATDESCRIPTOR 结构。这需要对 OpenGL 有一个较深的理解，并且要知道它在 Windows 上被执行的过程。我们首先采用其缺省值。随着编程的逐步深入，将会对它有一个更好的理解并能按自己的要求去进行设置。其缺省的设置如下：

```
PIXELFORMATDESCRIPTOR pfd =
{
```

```
    sizeof(PIXELFORMATDESCRIPTOR),          // size of this pfd
    1,                                      // version number
    PFD_DRAW_TO_WINDOW |                    // support window
      PFD_SUPPORT_OPENGL |                  // support OpenGL
    PFD_DOUBLEBUFFER,                       // double buffered
    PFD_TYPE_RGBA,                          // RGBA type
    24,                                     // 24-bit color depth
    0, 0, 0, 0, 0, 0,                       // color bits ignored
    0,                                      // no alpha buffer
    0,                                      // shift bit ignored
    0,                                      // no accumulation buffer
    0, 0, 0, 0,                             // accum bits ignored
    32,                                     // 32-bit z-buffer
    0,                                      // no stencil buffer
    0,                                      // no auxiliary buffer
    PFD_MAIN_PLANE,                         // main layer
    0,                                      // reserved
    0, 0, 0                                 // layer masks ignored
};
```

在这个设置中，nSize 置为 sizeof(PIXELFORMATDESCRIPTOR)，结构的版本号 nVersion通常被设置为 1。dwFlags 则是 PFD_DRAW_TO_WINDOW、PFD_SUPPORT_OPENGL 及 PFD_DOUBLEBUFFER 的逻辑或。iPixelType 像素类型，若采用 RGBA 颜色模式，则应设置为 PFD_TYPE_RGBA。随着显示设备的不断更新和图形加速性能的不断提高，很少使用颜色索引模式。对应于 PFD_TYPE_GRBA 设置，颜色位数可以为 24 位真彩色，这样可以显示 1670 万种不同的颜色。cAccumBits 被设为 0，则不使用累积缓存。cDepthBits 设为 24 位深度缓存，可以从物体中消除隐藏面。cStencilBits 为 0，表示不使用模板缓存。iLayerType 被设置成 PFD_MAIN_PLANE，表示选择主层面。不过，我们可以忽略它。

3.3.2 设置像素格式

一旦对结构 PIXELFORMATDESCRIPTOR 进行了初始化，就可以设置像素格式了。下面的代码段可以完成这项任务。

```
int iPixelformat;

if ( (iPixelformat = ChoosePixelFormat(m_pDC->GetSafeHdc(), &pfd)) == 0 )
{
    MessageBox("ChoosePixelFormat failed");
    return FALSE;
}

if (SetPixelFormat(m_pDC->GetSafeHdc(), iPixelformat, &pfd) == FALSE)
```

```
{
    MessageBox("SetPixelFormat failed");
    return FALSE;
}
```

上段代码中，ChoosePixelFormat()函数在硬件所支持的像素格式中选择与 pfd 所要求的像素格式最匹配的一个。该函数有两个参数，第一个是设备描述表的句柄，以便告诉 Windows 从设备描述表支持的像素格式中进行选择，第二个参数是所要求的像素格式的地址。如果这种选择成功，则返回匹配的像素格式的索引值，如果不成功，则返回 0。

SetPixelFormat()函数则把设备描述表的像素格式设置成由索引值 iPixelformat 所指向的像素格式。该函数有 3 个参数，第一个参数是设备描述表的句柄，第二个参数是像素格式的索引值，第三个参数为结构 PIXELFORMATDESCRIPTOR 的地址。如果该函数成功调用，则返回 GL_TRUE；否则返回 GL_FALSE。

必须指出的是，一个应用程序只能为一个窗口设置一次像素格式，并且一旦设置，在程序执行的过程中，就不能进行更改。而且，在创建着色描述表之前，必须设置像素格式，否则着色描述表就不知道它该选择什么样的像素属性了。

3.3.3 测试像素格式

对于大多数应用程序，通过 ChoosePixelFormat()函数所选择的像素格式一般不会引起问题。但是对像素格式进行测试以确保它适合自己的应用程序也是很重要的，因为它能保证应用程序在任何时候都不会出问题。获得想要的像素格式的一个最直接的方法就是对设备描述表所支持的每一个像素格式进行测试，从中选择与自己所想要的像素格式最匹配的一个。下面这段代码应用了这个方法。

```
HDC  hdc

// local variables
PIXELFORMATDESCRIPTOR    pfd;
int    Max,    PixelFormat ;

// initialize a pixel format index variable
iPixelFormat = 1;

// keep obtaining and examining pixel format data...
do {
    // try to obtain some pixel format data
    iMax = DescribePixelFormat(hdc, iPixelFormat, sizeof(pfd), &pfd);

    // if there was some problem with that...
    if (iMax == 0)

        // return indicating failure
```

```
            return(FALSE);

        // we have successfully obtained pixel format data
        // let's examine the pixel format data...
        myPixelFormatExaminer (&pfd);
        }
    // ...until we've looked at all the device context's pixel formats
    while (++iPixelFormat <= iMax);
```

在上述代码中,DescribePixelFormat()函数获得当前平台所支持的像素格式的最大数目。该函数有4个参数:设备描述表句柄、要测试的像素格式的索引值、结构 PIXELFORMATDESCRIPTOR 的大小及地址。该函数把与索引值 iPixelFormat 相对应的像素格式写入由 &pfd 指定的 PIXELFORMATDESCRIPTOR 结构。在获得了像素格式的最大数目之后,就进行循环。

myPixelFormatExaminer () 是自己的一个检测和修正器,以便按自己的方式进行决策。

为了灵活方便,在此本书并不给出 myPixelFormatExaminer ()的具体模式。在以后的编程中,可采用如下语句来代替上面的测试。

```
int         n;
n = ::GetPixelFormat(m_pDC->GetSafeHdc());
::DescribePixelFormat(m_pDC->GetSafeHdc(), n, sizeof(pfd), &pfd);
```

其中,GetPixelFormat()函数可获得当前像素格式的索引值。

3.4 创建着色描述表

一个 OpenGL 应用程序必须有一个着色描述表,并且在应用 OpenGL 绘制之前它就应该是当前的。有两种方法创建着色描述表。第一种方法是通过响应 Windows 消息 WM_CREATE 来创建并当前化一个着色描述表。通过响应 Windows 消息 WM_DESTROY 来使一个着色描述表不是当前的并删除它。第二种方法是通过响应 Windows 消息 WM_CREATE 来创建一个着色描述表,但只在每次应用 OpenGL 进行绘制之前才立即当前化,并且一旦此次绘制完成就立即使其非当前化。最后通过响应 Windows 消息 WM_DESTROY 来删除它。

每一种方法都有其优点和缺点。第一种方法的优点在于应用程序只当前化一次着色描述表,减少了许多开销,加快了程序的执行。缺点是必须在整个应用程序的运行过程中保留一个DC。相比较而言,第二种方法易于实现,但需要更多的执行时间。第一种方法可以按下述代码来建立:

```
int CStepinGlView::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    if (CView::OnCreate(lpCreateStruct) == -1)
        return -1;

    // TODO: Add your specialized creation code here
```

```
    // local variables
    HDC             hdc ;
    HGLRC           hglrc ;
    BOOL            bHaveCurrentRC;

    // obtain a device context for the window
    hdc = GetDC(hWnd);

    // set an appropriate pixel format
    myPixelFormatSetupFunction(hdc);

    // if we can create a rendering context ...
    if (hglrc = wglCreateContext( hdc ) ) {

        // try to make it the thread's current rendering context
        bHaveCurrentRC = wglMakeCurrent(hdc, hglrc) ;

        }

    // perform miscellaneous other WM_CREATE chores ...
}

HWND        hwnd;

void CStepinGlView::OnDestroy()
{

    // TODO: Add your message handler code here

    HGLRC    hglrc;
    HDC      hdc ;

    // if the thread has a current rendering context ...
    if(hglrc = wglGetCurrentContext()) {

      // obtain its associated device context
      hdc = wglGetCurrentDC() ;

      // make the rendering context not current
      wglMakeCurrent(NULL, NULL) ;

      // release the device context
      ReleaseDC (hwnd, hdc) ;
```

```
        // delete the rendering context
        wglDeleteContext(hglrc);

    CView::OnDestroy();
}
```

第二种方法类似如下代码：

```
class CStepinGlView : public CView
{
protected: // create from serialization only
    CStepinGlView();
    DECLARE_DYNCREATE(CStepinGlView)

// Attributes
public:
    CStepinGlDoc * GetDocument();
    ... ...

protected:
    HGLRC        m_hRC;
    ... ...
}

int CStepinGlView::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    if (CView::OnCreate(lpCreateStruct) == -1)
        return -1;

    // TODO: Add your specialized creation code here

    CClientDC clientDC(this);
    myPixelFormatSetupFunction(clientDC.m_hDC);
    m_hRC = wglCreateContext(clientDC.m_hRC);

    ... ...

    return 0;
}

void CStepinGlView::OnDraw(CDC * pDC)
{
    CStepinGlDoc * pDoc = GetDocument();
    ASSERT_VALID(pDoc);
```

```
    // TODO: add draw code for native data here

    wglMakeCurrent(pDC->m_hDC,m_hRC);
    DrawScene();
    wglMakeCurrent(pDC->m_hDC,NULL);
}

void CStepinGlView::OnDestroy()
{
    // TODO: Add your message handler code here

    wglDeleteContext(m_hRC);

    CView::OnDestroy();
}
```

在上述两段代码中，OnCreate()函数通过调用 wglCreateContext()函数来创建一个与给定的 DC 兼容的着色描述表。这个函数只有一个参数，即 DC 的句柄。如果调用成功，返回一个刚建立的着色描述表的句柄，如果不成功，则返回 NULL。

wglMakeCurrent(hdc,hglrc)及 wglMakeCurrent(pDC->m_hDC,m_hRC) 使着色描述表当前化。wglMakeCurrent()函数有两个参数，其一是 DC 的句柄，其二是着色描述表的句柄。如果成功调用就返回 GL_TRUE，否则返回 GL_FALSE。wglMakeCurrent(NULL, NULL)及 wglMakeCurrent(pDC->m_hDC,NULL)则使当前着色描述表不再是当前的。必须指出，它们完全是一样的。也就是说，只要第二个参数为 NULL，则一定可以使当前着色描述表非当前化，且第一个参数自动被忽略。

wglGetCurrentContext()函数可获得当前的着色描述表句柄。wglGetCurrentDC()函数则返回一个绑定上当前着色描述表的设备描述表句柄。两函数均无参数。

最后，当关闭应用程序时，调用视窗类消息函数 OnDestroy()。在 OnDestroy() 中，wglDeleteContext()函数删除着色描述表。该函数用着色描述表的句柄作为其参数，如果成功调用则返回 GL_TRUE，否则返回 GL_FALSE。在删除一个着色描述表之前必须使其非当前化。当删除了一个着色描述表之后，必须删除与着色描述表联系的设备描述表。在第二种方法中，由于 DC 的创建和删除是响应 Windows 消息 WM_PAINT 完成的，因此没有必要删除。

3.5 极小化 OpenGL

现在，读者已经对像素格式和着色描述表有了一个认识，并且也有了设置像素属性和创建着色描述表的材料。在编程的过程中，只需注意各部分之间的有机联系。可以按下面的步骤，建立极小化程序。如果在建立程序的过程中有问题，也可以参考本章后面的程序列表。如果问题是来自于 Visual C++，请参考相应的 Visual C++ 手册。

3.5.1 创建用户界面

请按如下步骤操作：

(1) 创建新的 Visual C++ 工程

启动 Visual C++，准备创建一个新的工程。选择 File/New。在弹出的对话框中单击 Projects 标签，选择 MFC AppWizard(exe)，在 Project Name 栏中输入 StepinGl，在 Location 中输入目录。在以后的创建步骤中，可以按下面的指示进行。

AppWizard Step_1：	Single document
AppWizard Step_2：	None
AppWizard Step_3：	None 并单击 ActiveX controls
AppWizard Step_4：	Use 3D Controls
AppWizard Step_5：	缺省设置
AppWizard Step_6：	缺省设置

(2) 编辑菜单

在 Workspace Window 中单击 Resource 标签，打开 menu 并双击 IDR_MAINFRAME，出现菜单编辑窗口。删除 File 菜单中除 Exit 的各菜单项，删除 Edit 菜单。

(3) 编辑对话框

在 Workspace Window 中单击 Resource 标签，打开 Dialog 并双击 IDD_ABOUTBOX，出现 About StepinGl 对话框编辑器。如图 3-1 所示。

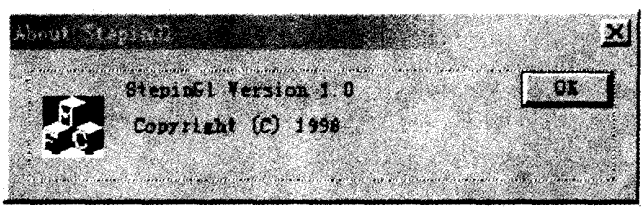

图 3-1

按自己的方式修改这个对话框。如图 3-2 所示。

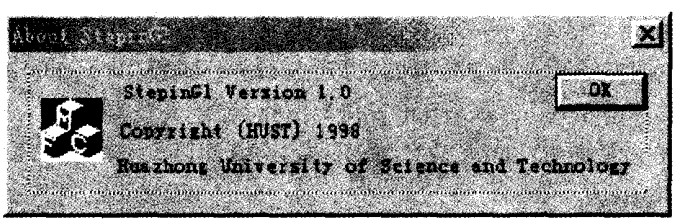

图 3-2

(4) 包含头文件与函数库文件

在 Workspace Window 中单击 File 标签,打开 Source Files,双击 StepinGlView.h 以打开此文件。在文件首部添加头文件:

```
#include "gl \ gl.h"
#include "gl \ glu.h"
```

Visual C++ 编译器通知应用程序将应用到 OpenGL 函数。因此,gl.h 文件是必不可少的。glu.h 则表示要用到 OpenGL 实用库函数。

打开菜单 Project \ Settings,在弹出的对话框中选择 Link 标签,在 Object \ Libaray Modules 栏中增加 OpenGl32.lib 及 glu32.lib 两个文件。为了访问 OpenGL 函数,必须把 OpenGL32.lib 文件加入到工程中去,该函数库文件包含一些重要信息,这些信息可帮助程序利用链接器访问动态链接库函数 OpenGL32.DLL。glu32.lib 则表示要用到 OpenGL 实用库函数。

(5) 编辑 MainFrm.cpp 文件

在 Workspace Window 中单击 File 标签,打开 Source Files,双击 MainFrm.cpp 打开此文件。编辑 PreCreateWindow()函数:

```
BOOL CMainFrame::PreCreateWindow(CREATESTRUCT& cs)
{
    // TODO: Modify the Window class or styles here by modifying
    // the CREATESTRUCT cs

    cs.cx = 300;
    cs.cy = 335;

    return CFrameWnd::PreCreateWindow(cs);
}
```

(6) 运行程序

编译连接并运行程序(按 F5 键)。正如读者所见,窗口中什么也没有。

3.5.2 极小化 OpenGL

第一步:设置像素格式和管理着色描述表

(1) 设置像素格式

在 Workspace Window 中单击 File 标签,打开 Source Files,双击 StepinGlView.cpp 打开此文件。在工具条中,单击 WizardBar Action 三角形图标,选择 Add Windows Message Hander 选项,在弹出的对话框中增加 Windows 消息 WM_CREATE,选择 Add and Edit 按钮为其生成消息响应函数 OnCreate()。

在工具条中,单击 WizardBar Action 三角形图标,选择 Add Member Function 选项,编辑 Add Member Function 对话框。Function Type 栏中输入 void,在 Function Declaration 中输入

Init(),选择 Protected。如图 3-3 所示。

按上述方法,增加保护成员函数 bSetupPixelFormat(),其返回类型为 BOOL。打开 StepinGlView.h 头文件,声明两个公有成员如下:

```
//声明两个公有成员
CClientDC *m_pDC;
CRect      m_oldRect;
```

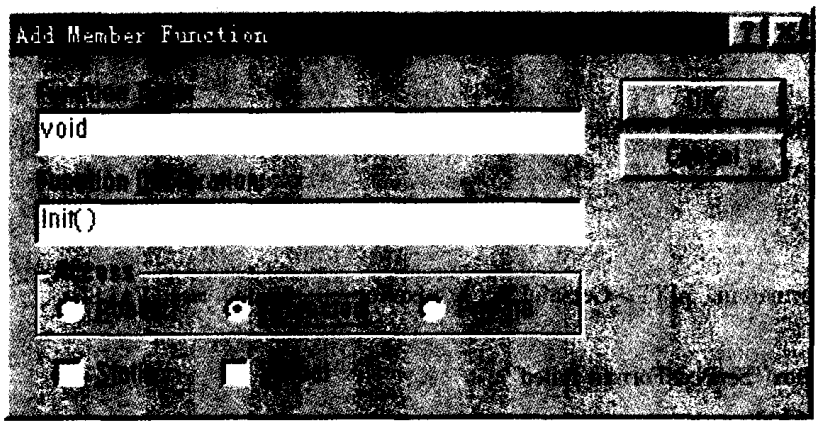

图 3-3

打开 StepinGlView.cpp 文件,在构造函数 CStepinGlView()中对 m_pDC 初始化。

```
CStepinGlView::CStepinGlView()
{
        // TODO: add construction code here
        m_pDC = NULL;
}
```

选择成员函数列表中的 bSetupPixelFormat,编辑该函数:

```
BOOL CStepinGlView::bSetupPixelFormat()
{
    static PIXELFORMATDESCRIPTOR pfd =
    {
      sizeof(PIXELFORMATDESCRIPTOR),        // size of this pfd
      1,                                    // version number
      PFD_DRAW_TO_WINDOW |                  // support window
        PFD_SUPPORT_OPENGL,                 // support OpenGL
      PFD_TYPE_RGBA,                        // RGBA type
      24,                                   // 24-bit color depth
      0, 0, 0, 0, 0, 0,                     // color bits ignored
      0,                                    // no alpha buffer
      0,                                    // shift bit ignored
      0,                                    // no accumulation buffer
      0, 0, 0, 0,                           // accum bits ignored
```

```
        32,                                    // 32-bit z-buffer
        0,                                     // no stencil buffer
        0,                                     // no auxiliary buffer
        PFD_MAIN_PLANE,                        // main layer
        0,                                     // reserved
        0, 0, 0                                // layer masks ignored
    };
    int pixelformat;

    if ( (pixelformat = ChoosePixelFormat(m_pDC->GetSafeHdc(), &pfd)) == 0 )
    {
        MessageBox("ChoosePixelFormat failed");
        return FALSE;
    }

    if (SetPixelFormat(m_pDC->GetSafeHdc(), pixelformat, &pfd) == FALSE)
    {
        MessageBox("SetPixelFormat failed");
        return FALSE;
    }

    return TRUE;
}
```

选择成员函数列表中的 Init,编辑该函数:

```
void CStepinGlView::Init()
{
    PIXELFORMATDESCRIPTOR pfd;
    int         n;
    HGLRC       hrc;

    m_pDC = new CClientDC(this);

    ASSERT(m_pDC != NULL);

    if (! bSetupPixelFormat()) return;
}
```

到此为止,就完成了像素格式的设置。

(2) 创建着色描述表并当前化着色描述表

根据前面介绍的方法,可在刚编辑的 Init()函数后追加以下代码:

```
    n = ::GetPixelFormat(m_pDC->GetSafeHdc());
    ::DescribePixelFormat(m_pDC->GetSafeHdc(), n, sizeof(pfd), &pfd);
```

```
hrc = wglCreateContext(m_pDC->GetSafeHdc());
wglMakeCurrent(m_pDC->GetSafeHdc(), hrc);
```

(3) 删除着色描述表

在工具条中,单击 WizardBar Action 三角形图标,选择 Add Windows Message Hander 选项,在弹出的对话框中增加 Windows 消息 WM_DESTROY,选择 Add and Edit 按钮为其生成消息响应函数 OnDestroy()。

下面的语句删除着色描述表以及绑定上该着色描述表的设备描述表。

```
void CStepinGlView::OnDestroy()
{
    // TODO: Add your message handler code here

    HGLRC hrc;

    hrc = ::wglGetCurrentContext();

    ::wglMakeCurrent(NULL, NULL);

    if (hrc)
      ::wglDeleteContext(hrc);

    if (m_pDC)
      delete m_pDC;

    CView::OnDestroy();
}
```

至此,已正确地创建了像素格式和着色描述表并对其进行了有效的管理。但是,必须通过响应 Windows 消息 WM_CREATE 才能使 OpenGL 内核得到通知。也就是说,必须在 OnCreate()函数中调用 Init()函数。这样 OnCreate()函数就变为如下形式:

```
int CStepinGlView::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    if (CView::OnCreate(lpCreateStruct) == -1)
        return -1;

    // TODO: Add your specialized creation code here

    Init();

    return 0;
}
```

(4) 设定 OpenGL 风格

选择成员函数列表中的 PreCreateWindow(),编辑该函数:

```
BOOL CStepinGlView::PreCreateWindow(CREATESTRUCT& cs)
{
    // TODO: Modify the Window class or styles here by modifying
    // the CREATESTRUCT cs

    cs.style |= WS_CLIPCHILDREN |WS_CLIPSIBLINGS;

    return CView::PreCreateWindow(cs);
}
```

OpenGL 有自己的像素格式,只有 OpenGL 的客户区才能接受 OpenGL 绘制,因此一般应有这两个风格设置。

(5) 运行程序

第二次编译连接并运行该程序。结果与前面一样,什么也没有。

第二步:建立 OpenGL 视点

(1) 编辑 Init()函数

选择成员函数列表中的 Init,在已编辑的 Init()函数后追加以下语句:

```
GetClientRect(&m_oldRect);
glClearDepth(1.0f);
glEnable(GL_DEPTH_TEST);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
```

此时,Init()函数为:

```
void CStepinGlView::Init()
{
    PIXELFORMATDESCRIPTOR pfd;
    int         n;
    HGLRC       hrc;

    m_pDC = new CClientDC(this);

    ASSERT(m_pDC != NULL);

    if (! bSetupPixelFormat())
        return;

    n = ::GetPixelFormat(m_pDC->GetSafeHdc());
    ::DescribePixelFormat(m_pDC->GetSafeHdc(), n, sizeof(pfd), &pfd);
```

```
    hrc = wglCreateContext(m_pDC->GetSafeHdc());
    wglMakeCurrent(m_pDC->GetSafeHdc(), hrc);

    GetClientRect(&m_oldRect);

    glClearDepth(1.0f);
    glEnable(GL_DEPTH_TEST);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}
```

上述语句的作用分别是获得客户区的大小、清除深度缓存、启动深度测试及启动 OpenGL 模型变换。第 5 章将介绍这些语句的含义。

(2) 添加 Windows 消息

在工具条中,单击 WizardBar Action 三角形图标,选择 Add Windows Message Hander 选项,在弹出的对话框中增加 Windows 消息 WM_SIZE,选择 Add and Edit 按钮为其生成消息响应函数 OnSize()。编辑如下:

```
void CStepinGlView::OnSize(UINT nType, int cx, int cy)
{
      CView::OnSize(nType, cx, cy);

      // TODO: Add your message handler code here

if(cy > 0)
    {
          if((m_oldRect.right > cx) || (m_oldRect.bottom > cy))
          RedrawWindow();

    m_oldRect.right = cx;
    m_oldRect.bottom = cy;

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
          glFrustum(-1.0,1.0,-1.0,1.0,0.0,7.0);
          glViewport(0, 0, cx, cy);
    }
}
```

上述代码主要用于建立视点、启动透视变换及建立透视体(架设了一架相机,只待按一下快门了!)。这些语句的含义在第 5 章中解释。

(3) 运行程序

第三次编译连接并运行程序。此时,窗口的用户区已变成黑色。

第三步:建立绘制函数 DrawScene()

(1) 增加保护成员函数

添加保护成员函数 DrawScene()。该函数具体绘制 OpenGL 图形。

(2) 编辑 OnDraw()

选择成员函数列表中的 OnDraw,并做如下编辑:

```
void CStepinGlView::OnDraw(CDC * pDC)
{
    CStepinGlDoc * pDoc = GetDocument();
    ASSERT_VALID(pDoc);

    // TODO: add draw code for native data here

    DrawScene();
}
```

DrawScene()函数是用来编写 OpenGL 绘制语句的。所有的 OpenGL 绘制工作都可以通过它来完成。今后,读者就可以把工作的重点放在怎样去设计自己的场景了,而不必去考虑程序与 Windows 发生联系的细节。至此,极小化 OpenGL 的工作就已经完成。很幸运,无论绘制多大多么复杂的场景,初始化 OpenGL 的工作都一样,而且仅此而已。

(3) 编辑 DrawScene()函数

选择成员函数列表中的 DrawScene,做如下编辑:

```
void CStepinGlView::DrawScene()
{
    glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glColor3f(1.0f, 0.0f, 0.0f);

    glBegin(GL_LINES);
        glVertex2f(-0.8f,0.8f);
        glVertex2f(0.8f,-0.8f);
    glEnd();
    glFinish();
}
```

(4) 运行程序

第四次编译连接并运行程序。此时的窗口的用户区已变成黑色,且显示了一条红色的直线。如图 3-4 所示。

3.6 OpenGL 颜色

本节并不属于 OpenGL 极小化的工作。但当颜色模式被指定为 RGBA 时,为一个物体指

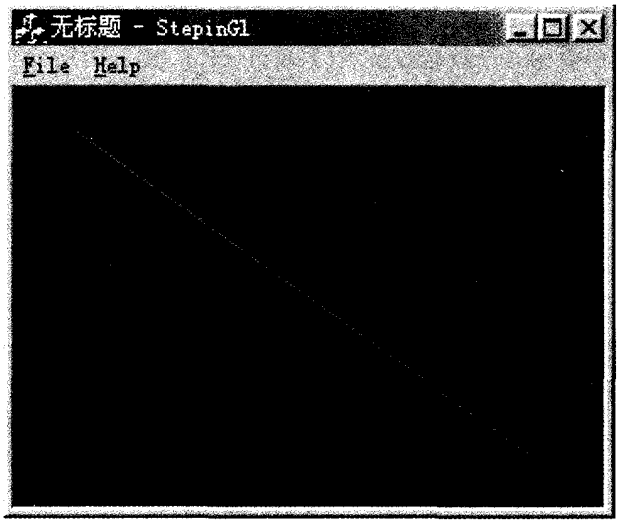

图 3-4　极小化 OpenGL

定颜色是相当容易的事。

现在以 DrawScene()函数中的语句为例,简单讨论这个问题。有关 OpenGL 的颜色问题请参考第 7 章。

```
void CStepinGlView::DrawScene()
{
    glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glColor3f(1.0f, 0.0f, 0.0f);

    glBegin(GL_LINES);
        glVertex2f(-0.8f,0.8f);
        glVertex2f(0.8f,-0.8f);
    glEnd();

    glFinish();
}
```

DrawScene()函数中的第一条语句调用 glClearColor()函数。该函数用来设置图像的背景颜色,它有 4 个参数,每个参数的数据类型为 GLclampf,依次表示红色、绿色、蓝色及 Alpha 成份。

当采用 RGBA 颜色模式时,颜色是由红色、绿色、蓝色 3 种颜色组合而成的。其颜色成份的表示方法与通常的 Windows 颜色表示方法不相同。它是用 0.0 ~ 1.0 之间的一个浮点值来表示的,而不是用通常采用的 0 ~ 255 之间的一个整数来表示。值为 0.0 时,表示最后的颜色中没有该分量颜色的成份;值为 1.0 时,表示取该分量颜色的最大可能的浓度。glClearColor()函数用(0.0f,0.0f,0.0f,1.0f)把背景的颜色设置成黑色。该函数的第四个参数为 Alpha 参

数,主要说明颜色的透明性,可用它来产生如融合等特殊效果。通常地,Alpha 值对背景的颜色没有什么影响。该函数的 4 个缺省参数为(1.0f,1.0f,1.0f,1.0f)。

在 DrawScene()函数中,第二条语句调用 glClear()函数。该函数用标识 GL_COLOR_BUFFER_BIT 通知 OpenGL 用当前设置的背景颜色第一次清除颜色缓存。颜色缓存事实上是一块内存区,它保存了将在屏幕上显示的图像的颜色信息。glClear()函数还用标识 GL_DEPTH_BUFFER_BIT 清除深度缓存,这样以便 OpenGL 进行深度测试和消除隐藏面。注意,该函数还可以用其他的标识操作相应的缓存。

glColor3f()函数设定当前的物体绘制颜色。该函数有 3 个类型为 GLfloat 的参数,依次表示该颜色的红色、绿色、蓝色成份,共有 32 种不同的形式。一旦用 glColor()函数设定了物体的绘制颜色,以后 OpenGL 所绘制的所有物体的颜色均为该种颜色,直到用不同的参数再次调用该函数。今后还会看到,一个物体的颜色会极大地受到其他 OpenGL 图形操作的影响。

3.7 程序清单

StepinGL 应用程序包含很多文件,这里只列出被我们修改过的文件。目前只涉及到两个类 3 个文件,即:CStepinGlView 和 CMainFrame 类,StepinGlView.h、StepinGlView.cpp 和 MainFrm.cpp 文件。其他所有文件是在建立 StepinGl 应用程序时由 AppWizard 建立的。

本章是以后各章的基础,因此,我们在此列出程序清单。后面的章节将不列出这样的程序清单,因为,只要读者按书中步骤进行就能完成编程工作。

StepinGlView.h 文件:

```
//////////////////////////////////////////////////////////////////////////////////////
//////////////////////////                                  //////////////////////////
//////////////////////////        StepinGlView.h :          //////////////////////////
//////////////////////////                                  //////////////////////////
//////////////////////////////////////////////////////////////////////////////////////

#include "gl\gl.h"
#include "gl\glu.h"

#if !defined(AFX_STEPINGLVIEW_H__4D9DCF8F_170C_11D2_83B2_0080C832FCF3__INCLUDED_)
#define AFX_STEPINGLVIEW_H__4D9DCF8F_170C_11D2_83B2_0080C832FCF3__INCLUDED_

#if _MSC_VER >= 1000
#pragma once
#endif // _MSC_VER >= 1000

class CStepinGlView : public CView
{
protected: // create from serialization only
```

```
        CStepinGlView();
        DECLARE_DYNCREATE(CStepinGlView)

// Attributes
public:
        CStepinGlDoc * GetDocument();

        CClientDC      * m_pDC;
        CRect          m_oldRect;

// Operations
public:

// Overrides
        // ClassWizard generated virtual function overrides
        //{{AFX_VIRTUAL(CStepinGlView)
        public:
        virtual void OnDraw(CDC * pDC); // overridden to draw this view
        virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
        protected:
        //}}AFX_VIRTUAL

// Implementation
public:
        void DrawScene();
        BOOL bSetupPixelFormat();
        void Init();
        virtual CStepinGlView();
#ifdef _DEBUG
        virtual void AssertValid() const;
        virtual void Dump(CDumpContext& dc) const;
#endif

protected:

// Generated message map functions
protected:
        //{{AFX_MSG(CStepinGlView)
        afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
        afx_msg void OnSize(UINT nType, int cx, int cy);
        afx_msg void OnDestroy();
        //}}AFX_MSG
        DECLARE_MESSAGE_MAP()
};
```

```
#ifndef _DEBUG // debug version in StepinGlView.cpp
inline CStepinGlDoc * CStepinGlView::GetDocument()
   { return (CStepinGlDoc *)m_pDocument; }
#endif

/////////////////////////////////////////////////////////////////////////////
//{{AFX_INSERT_LOCATION}}
// Microsoft Developer Studio will insert additional declarations immediately before the previous line.

#endif // !defined(AFX_STEPINGLVIEW_H__4D9DCF8F_170C_11D2_83B2_0080C832FCF3__INCLUDED_)
```

StepinGlView.pp 文件:

```
/////////////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////                                   ////////////////////////////
////////////////////////////          StepinGlView.cpp         ////////////////////////////
////////////////////////////                                   ////////////////////////////
/////////////////////////////////////////////////////////////////////////////////////////////////

#include "stdafx.h"
#include "StepinGl.h"

#include "StepinGlDoc.h"
#include "StepinGlView.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
// CStepinGlView

IMPLEMENT_DYNCREATE(CStepinGlView, CView)

BEGIN_MESSAGE_MAP(CStepinGlView, CView)
    //{{AFX_MSG_MAP(CStepinGlView)
    ON_WM_CREATE()
    ON_WM_SIZE()
    ON_WM_DESTROY()
    //}}AFX_MSG_MAP
```

```
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CStepinGlView construction/destruction

CStepinGlView::CStepinGlView()
{
	// TODO: add construction code here
	m_pDC = NULL;
}

CStepinGlView::CStepinGlView()
{
}

BOOL CStepinGlView::PreCreateWindow(CREATESTRUCT& cs)
{
	// TODO: Modify the Window class or styles here by modifying
	// the CREATESTRUCT cs
	cs.style |= WS_CLIPCHILDREN | WS_CLIPSIBLINGS;

	return CView::PreCreateWindow(cs);
}

/////////////////////////////////////////////////////////////////////////////
// CStepinGlView drawing

void CStepinGlView::OnDraw(CDC * pDC)
{
	CStepinGlDoc * pDoc = GetDocument();
	ASSERT_VALID(pDoc);

	// TODO: add draw code for native data here
	DrawScene();
}

/////////////////////////////////////////////////////////////////////////////
// CStepinGlView diagnostics

#ifdef _DEBUG
void CStepinGlView::AssertValid() const
{
	CView::AssertValid();
}
```

```
void CStepinGlView::Dump(CDumpContext& dc) const
{
        CView::Dump(dc);
}

CStepinGlDoc * CStepinGlView::GetDocument() // non-debug version is inline
{
        ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(CStepinGlDoc)));
        return (CStepinGlDoc *)m_pDocument;
}
#endif //_DEBUG

/////////////////////////////////////////////////////////////////////////////
// CStepinGlView message handlers

void CStepinGlView::Init()
{
    PIXELFORMATDESCRIPTOR pfd;
    int           n;
      HGLRC         hrc;

    m_pDC = new CClientDC(this);

    ASSERT(m_pDC != NULL);

    if (! bSetupPixelFormat())
        return;

    n = ::GetPixelFormat(m_pDC->GetSafeHdc());
    ::DescribePixelFormat(m_pDC->GetSafeHdc(), n, sizeof(pfd), &pfd);

    hrc = wglCreateContext(m_pDC->GetSafeHdc());
    wglMakeCurrent(m_pDC->GetSafeHdc(), hrc);

    GetClientRect(&m_oldRect);
    glClearDepth(1.0f);
    glEnable(GL_DEPTH_TEST);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

BOOL CStepinGlView::bSetupPixelFormat()
{
```

```
    static PIXELFORMATDESCRIPTOR pfd =
    {
      sizeof(PIXELFORMATDESCRIPTOR),
      1,
      PFD_DRAW_TO_WINDOW |
        PFD_SUPPORT_OPENGL,
      PFD_TYPE_RGBA,
      24,
      0, 0, 0, 0, 0, 0,
      0,
      0,
      0,
      0, 0, 0, 0,
      32,
      0,
      0,
      PFD_MAIN_PLANE,
      0,
      0, 0, 0
    };
    int pixelformat;

    if ( (pixelformat = ChoosePixelFormat(m_pDC->GetSafeHdc(), &pfd)) == 0 )
    {
        MessageBox("ChoosePixelFormat failed");
        return FALSE;
    }

    if (SetPixelFormat(m_pDC->GetSafeHdc(), pixelformat, &pfd) == FALSE)
    {
        MessageBox("SetPixelFormat failed");
        return FALSE;
    }

    return TRUE;
}

int CStepinGlView::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    if (CView::OnCreate(lpCreateStruct) == -1)
        return -1;

    // TODO: Add your specialized creation code here
    Init();
```

```
    return 0;
}

void CStepinGlView::DrawScene()
{
    glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glColor3f(1.0f, 0.0f, 0.0f);

    glBegin(GL_LINES);
        glVertex2f(-0.8f,0.8f);
        glVertex2f(0.8f,-0.8f);
    glEnd();

  glFinish();
}

void CStepinGlView::OnSize(UINT nType, int cx, int cy)
{
    CView::OnSize(nType, cx, cy);

    // TODO: Add your message handler code here
      if(cy > 0)
  {
        if((m_oldRect.right > cx) || (m_oldRect.bottom > cy))
        RedrawWindow();

      m_oldRect.right = cx;
      m_oldRect.bottom = cy;

      glMatrixMode(GL_PROJECTION);
      glLoadIdentity();
        glFrustum(-1.0,1.0,-1.0,1.0,1.0,7.0);
        glViewport(0, 0, cx, cy);
  }
}

void CStepinGlView::OnDestroy()
{
    // TODO: Add your message handler code here
    HGLRC hrc;
```

```
        hrc = ::wglGetCurrentContext();

    ::wglMakeCurrent(NULL, NULL);

    if (hrc)
        ::wglDeleteContext(hrc);

    if (m_pDC)
        delete m_pDC;

      CView::OnDestroy();
}
```

MainFrm.cpp 文件：

```
////////////////////////////////////////////////////////////////////////////////
//////////////////////////                              //////////////////////////
//////////////////////////          MainFrm.cpp         //////////////////////////
//////////////////////////                              //////////////////////////
////////////////////////////////////////////////////////////////////////////////

#include "stdafx.h"
#include "StepinGl.h"

#include "MainFrm.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
// CMainFrame

IMPLEMENT_DYNCREATE(CMainFrame, CFrameWnd)

BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)
    //{{AFX_MSG_MAP(CMainFrame)
        // NOTE - the ClassWizard will add and remove mapping macros here.
        //    DO NOT EDIT what you see in these blocks of generated code !
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
```

```
// CMainFrame construction/destruction

CMainFrame::CMainFrame()
{
	// TODO: add member initialization code here
}

CMainFrame::CMainFrame()
{
}

BOOL CMainFrame::PreCreateWindow(CREATESTRUCT& cs)
{
	// TODO: Modify the Window class or styles here by modifying
	// the CREATESTRUCT cs

	cs.cx = 300;
	cs.cy = 335;

	return CFrameWnd::PreCreateWindow(cs);
}

/////////////////////////////////////////////////////////////////////////////
// CMainFrame diagnostics

#ifdef _DEBUG
void CMainFrame::AssertValid() const
{
	CFrameWnd::AssertValid();
}

void CMainFrame::Dump(CDumpContext& dc) const
{
	CFrameWnd::Dump(dc);
}

#endif //_DEBUG

/////////////////////////////////////////////////////////////////////////////
// CMainFrame message handlers
```

第4章　绘制二维图形

第2章已经指出，应用OpenGL可以绘制点、线及多边形。由这些基本图形就可以建立复杂的三维模型。本章将介绍如何定义二维形状。学完本章之后，从方法上讲，读者可以完成任何复杂的三维物体的几何描述。

4.1　绘制点

在计算机图形学中，描述一个物体有许多方法，每一种方法都面临着如何表示物体的拓扑结构。作为3D的API，OpenGL也需要这种结构，但是其解决问题的重点在于怎样对物体进行数学描述，怎样组织好如此众多的OpenGL函数和功能来绘制图形。

OpenGL提供了描述点、线、多边形的绘制机制。它们通过glBegin()函数与glEnd()函数配对来完成。glBegin()函数有一个类型为GLenum的参数，它只能是如下10个常量中的一个：GL_POINTS，GL_LINES，GL_LINE_STRIP，GL_LINE_LOOP，GL_TRIANGLES，GL_TRIANGLE_STRIP，GL_TRIANGLE_FAN，GL_QUADS，GL_QUAD_STRIP，GL_POLYGON。

glEnd()函数则标志着定义形状的结束，该函数没有参数。glBegin()函数与glEnd()函数必须配对使用。

在OpenGL中，一个点是用其坐标来描述的，用glVertex()函数来完成其计算机图形学表示。

当glBegin()函数的参数为GL_POINTS时，表示在glBegin()/glEnd()函数对间将绘制点。可以像以下代码那样创建一系列的点：

```
glBegin(GL_POINTS);
    glVertex2f(0.0f,0.0f);
    glVertex2f(0.5f,0.5f);
    glVertex2f(-0.5f,0.5f);
    glVertex2f(0.5f,-0.5f);
    glVertex2f(-0.5f,-0.5f);
glEnd();
```

上述代码中，OpenGL把每个表示顶点的glVetex()函数解释为单个的点，这样上段代码便创建了5个独立的点。可以在glBegin()/glEnd()函数对中定义任意数目的点。上面定义的点是二维的。如果需要定义三维的点，则可以采用glVertex3f()等其他形式的函数：

```
glBegin(GL_POINTS);
    glVertex3f(0.0f,0.2f,0.3f);
    glVertex3f(0.5f,0.5f,-0.5f);
glEnd();
```

必须指出，glVetex()函数定义的点只具有数学意义。在屏幕上显示的点往往都比较大，这主要来自于两个方面，第一，浮点运算的精度是有限的，这就决定了点的坐标存在误差的可能性；第二，显示设备的分辨率是有限的。一个显示设备是用像素来显示图像的。一个点或由点构成的线最后都将进行离散化、整数化、设备坐标化。而一个像素是有大小的，这就决定了实际显示的点与数学意义上的点是有差别的。为了克服这些困难，OpenGL 作了一些优化处理，它采用 glPointSize()函数来自定义一个点在屏幕上的显示大小(注意还可以对一个点进行反走样处理)，例如：

```
glPointSize(2.0f);
```

参数 2.0 表示用 2 个像素来表示一个点的直径。

为了获取被设备支持的点的大小范围，OpenGL 提供了 glGet()函数。采用如下两条语句即可获得点的大小范围：

```
GLfloat     fPointSize[2];
glGetFloatv(GL_PIONT_SIZE_RANGE,fPointSize);
```

在这里，fPointSize 是一个有两个元素的数组。glGetFolatv()函数的第一个参数必须是 OpenGL 常量，第二个参数是一个数组的地址。函数按第一个参数的指示向第二个参数所指的地址传输数据。fPointSize[0]存储点的大小的最小值，fPointSize[1]则存储最大值。

当前点的大小也是可以获取的。采用如下所示语句：

```
GLfloat     fCurrentPointSize;
glGetFloatv(GL_POINT_SIZE, &fCurrentPointSize);
```

对应于 GL_POINT_SIZE 常量，fCurrentPointSize 参数是一个单变量的地址，用来记录当前点的大小。

下面的函数绘制了 5 个点，其结果如图 4-1 所示。

```
void DrawPoints()
{
    GLfloat fPointSize[2];
    glGetFloatv(GL_POINT_SIZE_RANGE,fPointSize);
    glPointSize(fPointSize[1]/2.0f);

    glBegin(GL_POINTS);
        glVertex2f(0.0f,0.0f);
        glVertex2f(0.5f,0.5f);
        glVertex2f(-0.5f,0.5f);
        glVertex2f(0.5f,-0.5f);
        glVertex2f(-0.5f,-0.5f);
    glEnd();
}
```

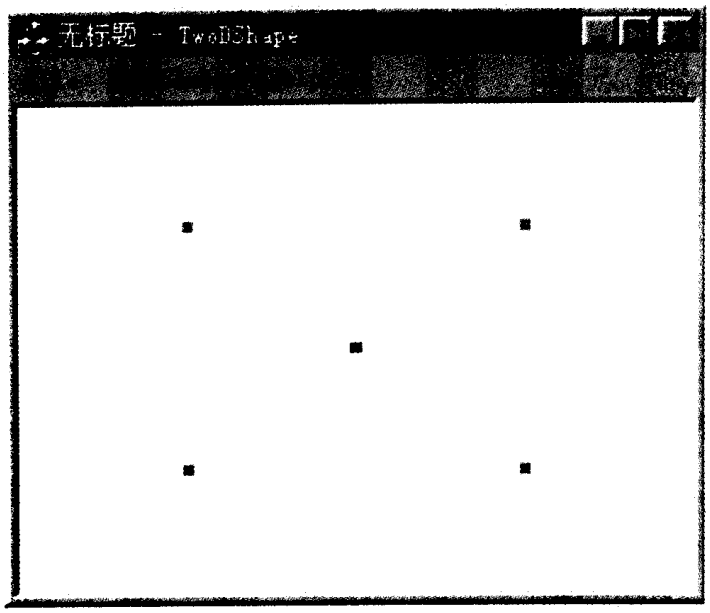

图 4-1　绘制点

4.2　绘制直线

4.2.1　绘制直线

一条直线由两点来定义。只要将 glBegin()函数的参数设为 GL_LINES,OpenGL 便可以用 glBegin()/glEnd()函数对来绘制直线。此时,OpenGL 则把 glBegin()/glEnd()函数对中的第一个顶点解释为第一条直线的起点,把第二个顶点解释为该直线的终点,第三个顶点则被解释为第二条直线的第一个顶点,依次类推。在 glBegin()/glEnd()函数对中,可以定义任意数目的直线。比如:

```
glBegin(GL_LINES);
    glVertex2f(0.5f,0.5f);
    glVertex2f(-0.50f,0.0f);
    glVertex2f(-0.5f,0.5f);
    glVertex2f(0.0f,-0.50f);
glEnd();
```

则可绘制两条直线。与对点的操作一样,glLineWidth()函数可定义直线的宽度:

```
glLineWidth(2.0f);
```

该函数申明了 2 个像素作为直线的宽度。为了确定线宽的范围,可采用以下语句:

```
GLfloat    fLineWidth[2];
glGetFloatv(GL_LINE_WIDTH_RANGE,fLineWidth);
```

其中，数组 fLineWidth 用来记录线宽的最小值和最大值。下述语句可获得当前的线宽：

```
GLfloat     fCurrentLineWidth;
glGetFloatv(GL_LINE_WIDTH, & fCurrentLineWidth);
```

下面的 DrawLines()函数绘制一条直线，其结果如图 4-2。

```
void CTwoDShapeView::DrawLines()
{
    GLfloat fLineWidth[2];
    glGetFloatv(GL_LINE_WIDTH_RANGE,fLineWidth);
    glLineWidth(fLineWidth[0]);

    glBegin(GL_LINES);
        glVertex2f(0.5f,0.5f);
        glVertex2f(-0.50f,0.0f);
    glEnd();
}
```

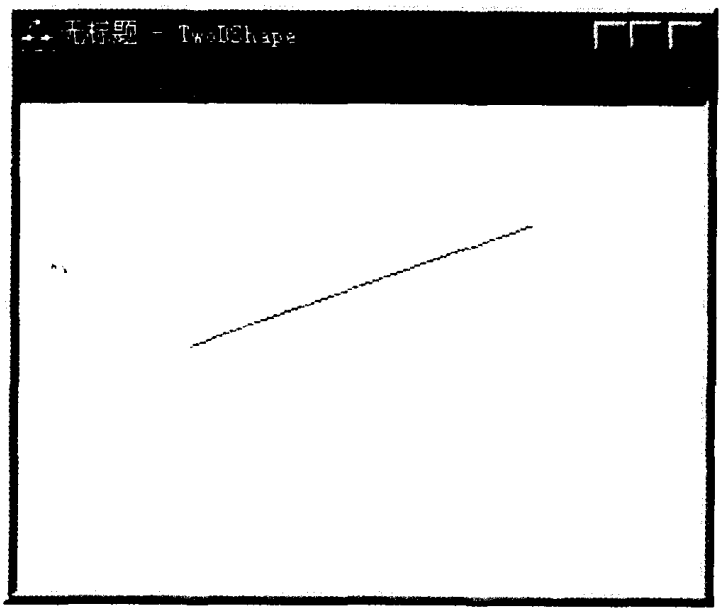

图 4-2　绘制直线

4.2.2 线　　型

OpenGL 不仅能画实线，而且可以绘制点画线(Stippled Lines)等其他类型的直线。为了绘制点画线，必须先自定义点画线的线型，然后通过 glLineStipple()函数让 OpenGL 获取所定义的线型。线型是通过二进制来描述的，1 代表一个实点，0 则表示在该像素处不画直线，即不填充像素。例如：

0000000011111111：　　表示空白线和长折线；

0000111100001111：　　表示空白线及短折线；

0000000100000001：　　表示点；

0001110001000111：　　表示折线点折线。

为了便于表示，一般将二进制表示成 16 进制。上述四个二进制数的 16 进制表示依次是 0x00FF，0x0F0F，0x0101，0x1c47。二进制按由低到高的顺序被处理。

一旦定义了线型，就应该调用 glLineStipple()函数让 OpenGL 知道已改变的线型。该函数有两个参数，第一个参数数据类型为 GLint，表示线型模式重复因子；第二个参数数据类型为 GLushort，表示所定义的线型。重复因子说明了用二进制表示的点的重复次数。如果重复因子是 2，一个 01010101 的线型模式事实上被当成 0011001100110011 线型来处理。

为了对所定义的线型获得 OpenGL 的支持，必须事先启动改变线型的机制：

```
glEnable(GL_LINE_STIPPLE);
```

在 OpenGL 中，glEnable()函数是一个重要的函数，通过该函数可以启动 OpenGL 的许多特定功能。该函数采用一个参数，这些参数均为 OpenGL 所定义的常量。大约有 50 个常量可供该函数使用。DrawLineStipple()函数的运行结果如图 4-3 所示。

```
void DrawLineStipple()
{
    GLfloat fLineWidth[2];
    glGetFloatv(GL_LINE_WIDTH_RANGE,fLineWidth);
    glLineWidth(fLineWidth[1]/6.0f);

    glBegin(GL_LINES);
        glVertex2f(-0.8f,0.8f);
        glVertex2f(0.80f,0.8f);
    glEnd();

    glEnable(GL_LINE_STIPPLE);

    glLineStipple(1,0x00FF);
    glBegin(GL_LINES);
        glVertex2f(-0.8f,0.4f);
        glVertex2f(0.80f,0.4f);
    glEnd();

    glLineStipple(1,0x0F0F);
    glBegin(GL_LINES);
        glVertex2f(-0.8f,0.0f);
        glVertex2f(0.80f,0.0f);
    glEnd();

    glLineStipple(2,0x0101);
    glBegin(GL_LINES);
        glVertex2f(-0.8f,-0.4f);
```

```
    glVertex2f(0.80f, -0.4f);
glEnd();

glLineStipple(1,0x1C47);
glBegin(GL_LINES);
    glVertex2f(-0.8f, -0.8f);
    glVertex2f(0.80f, -0.8f);
glEnd();

glDisable(GL_LINE_STIPPLE);
```

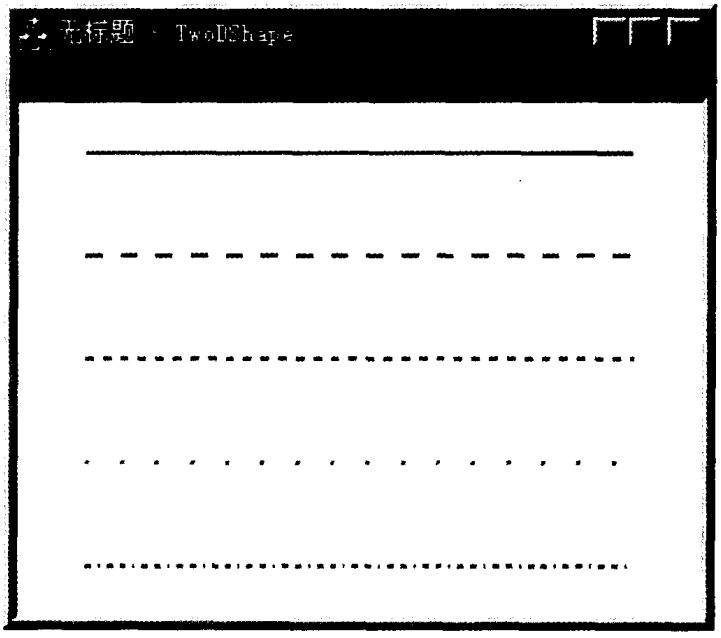

图 4-3 绘制线型

要注意,当不再使用自定义线型时,应将该功能关闭:

```
glDisable(GL_LINE_STIPPLE);
```

4.2.3 绘制折线

为了描述一条折线(Line strips),必须把 glBegin()函数的参数设为 GL_LINE_STRIP,然后在 glBegin()/glEnd()函数对间填写该折线的顶点。第一对顶点描述了折线的第一条线段,以后每增加一个顶点就增加一条线段。这与通常的 Windows GDI 绘制模式下采用 MoveTo()函数及 LineTo()函数绘制直线的方法是一致的。DrawLineStrip()函数绘制结果如图 4-4 所示。

```
void DrawLineStrip()
{
```

```
    GLfloat fLineWidth[2];
    glGetFloatv(GL_LINE_WIDTH_RANGE,fLineWidth);
    glLineWidth(fLineWidth[1]/6.0f);

    glBegin(GL_LINE_STRIP);
        glVertex2f(-0.8f,0.6f);
        glVertex2f(0.8f,0.6f);

        glVertex2f(-0.8f,0.2f);
        glVertex2f(0.8f,0.2f);
        glVertex2f(-0.8f,-0.2f);
        glVertex2f(0.8f,-0.2f);
        glVertex2f(-0.8f,-0.6f);
        glVertex2f(0.8f,-0.6f);
    glEnd();
}
```

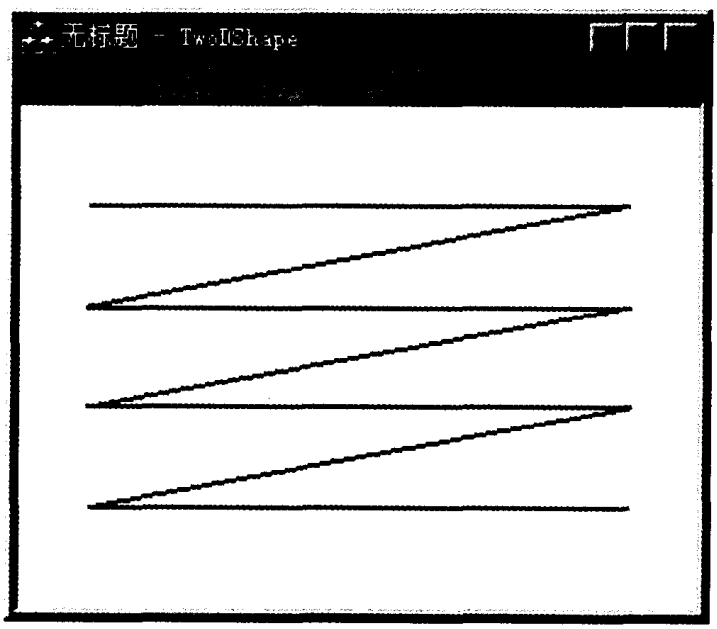

图 4-4　绘制折线

4.2.4　绘制闭合折线

绘制闭合折线(Line loops)的方法与绘制折线的方法完全一致,只是 glBegin()函数的参数必须为 GL_LINE_LOOP。二者唯一的区别在于最后一个顶点和第一个顶点要连上线段构成封闭的折线。DrawLineLoop()函数绘制一条闭合折线,如图 4-5 所示。

```
void DrawLineLoop()
{
    GLfloat fLineWidth[2];
```

```
glGetFloatv(GL_LINE_WIDTH_RANGE,fLineWidth);
glLineWidth(fLineWidth[1]/6.0f);

glBegin(GL_LINE_LOOP);
    glVertex2f(-0.8f,0.6f);
    glVertex2f(0.8f,0.6f);

    glVertex2f(-0.8f,0.2f);
    glVertex2f(0.8f,0.2f);
    glVertex2f(-0.8f,-0.2f);
    glVertex2f(0.8f,-0.2f);
    glVertex2f(-0.8f,-0.6f);
    glVertex2f(0.8f,-0.6f);
glEnd();
```

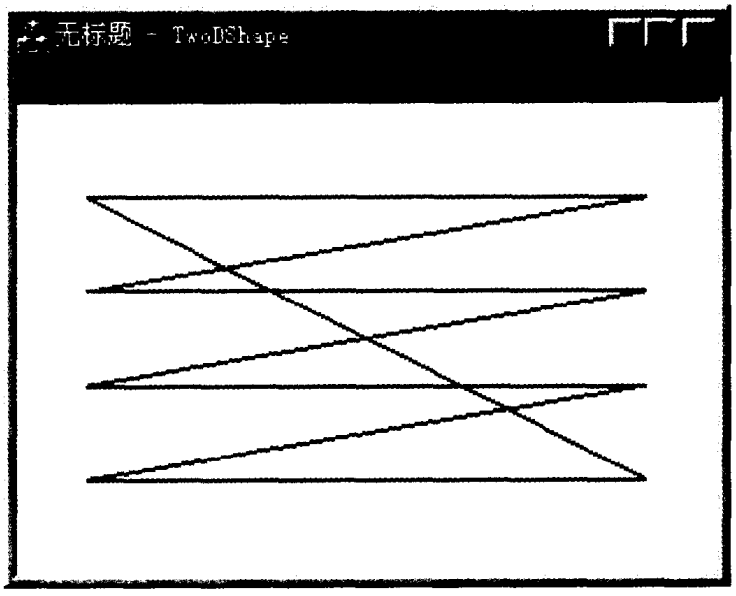

图 4-5　绘制闭合折线

4.3　绘制多边形

4.3.1　绘制多边形

在 OpenGL 中,一个多边形(Polygon)至少有 3 个顶点。直线不能相交,多边形应构成单连通的凸区域。一个多边形有前面和后面之分,并且前面和后面可以有不同的属性。比如,可以指定一个多边形的前面为红色而其后面为蓝色。在二维平面中,这似乎没有多大的意义,但是对于三维空间,却是十分重要的。比如,可以转动一个多边形以看到它的另一面。如图 4-6

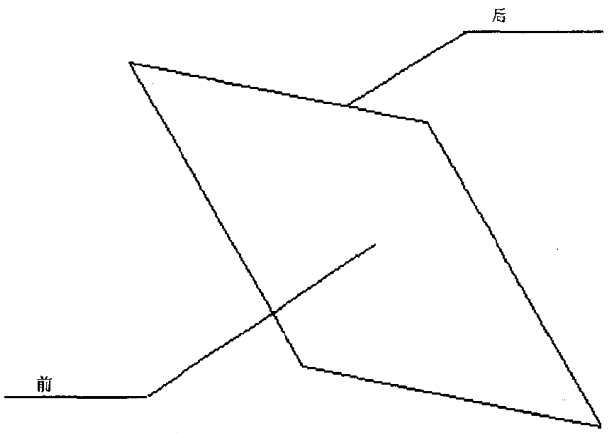

图 4-6　多边形面的区分

示。对一个面进行光照时，前后面的区分也是很重要的。

一个多边形的前面和后面是需要指定的，可以通过定义其顶点顺序来确定。缺省地，如果顶点的排列顺序是逆时针的，则该多边形是前面的；如果顶点的排列顺序是顺时针的，则该多边形是后面的。事实上，这是可以改变的。OpenGL 提供了一个灵活的函数：glFrontFace()。该函数有一个参数，且只能是 GL_CW 或 GL_CCW 中的一个。GL_CW 表示按顺时针顺序定义的多边形是前面的。GL_CCW 则表示按逆时针顺序定义的多边形是前面的，GL_CCW 是这个函数的缺省参数。另外，还可以通过计算多边形面积的符号来区分前后面，其图形学算法在此不作介绍。

由于要区分前面和后面，这就给多边形的定义带来了一些不灵活性，因为必须明确应采用什么样的顺序来按排其顶点。尽管如此，这种区分会给我们带来许多便利。比如，一个盒子内部看起来就应该和外部不一样；一张纸有前面和后面之分。还记得我们在前面提到过 3D Studio MAX 材质编辑器里有双面材质和双面光照吗？

多边形是封闭的线段构造的形状，它与用 GL_LINE_LOOP 绘制的闭合折线有本质的区别。一个多边形可以进行填充，也可以不填充。这样，就必须对多边形的绘制模式作出说明。可以采用 glPolygonMode()函数。该函数有两个参数，第一个参数可以是 GL_FRONT、GL_BACK、GL_FRONT_AND_BACK 中的一个，用来指定多边形的前面、后面或前后两面。第二个参数用来说明被指定面的绘制模式。GL_POINT 告诉 OpenGL 只在多边形的顶点画点；GL_LINE 则告诉 OpenGL 只画多边形的框架；而 GL_FILL 告诉 OpenGL 要填充多边形。

要定义一个多边形，只需将 glBegin()函数的参数设为 GL_POLYGON 即可。下面的代码定义一个四边形：

```
glBegin(GL_POLYGON);
    glVertex2f(-0.3f,0.3f);
    glVertex2f(-0.7f,-0.20f);
    glVertex2f(-0.3f,-0.40f);
    glVertex2f(0.4f,-0.40f);
    glVertex2f(0.8f,-0.2f);
    glVertex2f(0.4f,0.3f);
```

```
    glEnd();
```

上述代码中，多边形的顶点以逆时针顺序排列。如果 glFrontFace()函数采用缺省设置，则该多边形是前面的。

如果 glBegin()/glEnd()函数对间的顶点少于 3 个，则不会画多边形。下面的代码定义了一个六边形。如图 4-7 所示。

```
void DrawPolygons()
{
    glLineWidth(2.0f);
    glPolygonMode(GL_FRONT,GL_LINE);
    glPolygonMode(GL_BACK,GL_FILL);

    glFrontFace(GL_CCW);

    glBegin(GL_POLYGON);
        glVertex2f(-0.3f,0.3f);
        glVertex2f(-0.7f,-0.20f);
        glVertex2f(-0.3f,-0.40f);
        glVertex2f(0.4f,-0.40f);
        glVertex2f(0.8f,-0.2f);
        glVertex2f(0.4f,0.3f);
    glEnd();
}
```

上述代码中，glPolygonMode()函数被调用了两次。请思考这个问题，改变它们的参数，看看会产生什么结果。

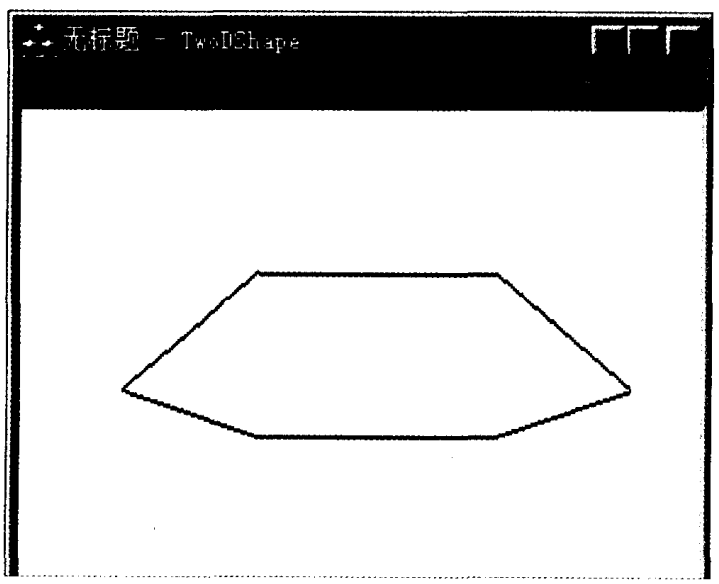

图 4-7　绘制多边形

4.3.2 多边形填充模式

正如可以自定义线型一样，OpenGL 也可以让用户自定义多边形填充模式。该填充模式应是一个 32 × 32 的位图，其中的每一个位代表屏幕上的一个点。位图则被 OpenGL 存储在内存中，在光栅化阶段把它取出，形成一个填充模式。

图 4-8 所示，便是一个位图网格模式，其中被填充位的值为 1，未被填充的值为 0。一般地，我们要将填充模式表示成 16 进制。在表示该位图时，以一个字节为单位，以 GLubyte 为数据类型，从左下角开始自左向右自下而上地表示。

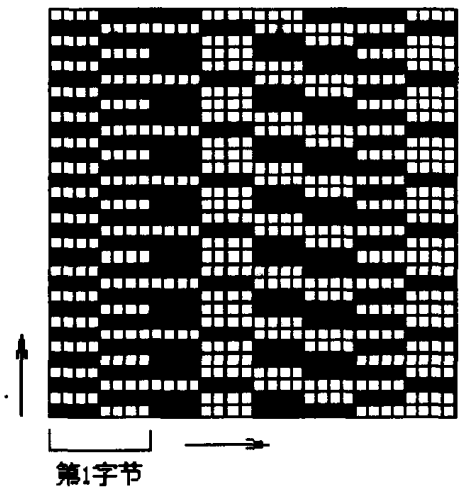

图 4-8 多边形填充位图

上述位图可表示成：

```
Glubyte     polygonFillPattern [] =
{
    0xF0, 0xF0, 0xFF, 0x00, 0x0F, 0xF0, 0xF0, 0xF0,
    0xF0, 0x0F, 0x00, 0x0F, 0x0F, 0xF0, 0x0F, 0xF0,
    0xF0, 0xF0, 0xFF, 0x00, 0x0F, 0xF0, 0xF0, 0xF0,
    0xF0, 0x0F, 0x00, 0x0F, 0x0F, 0xF0, 0x0F, 0xF0,
    0xF0, 0xF0, 0xFF, 0x00, 0x0F, 0xF0, 0xF0, 0xF0,
    0xF0, 0x0F, 0x00, 0x0F, 0x0F, 0xF0, 0x0F, 0xF0,
    0xF0, 0xF0, 0xFF, 0x00, 0x0F, 0xF0, 0xF0, 0xF0,
    0xF0, 0x0F, 0x00, 0x0F, 0x0F, 0xF0, 0x0F, 0xF0,
    0xF0, 0xF0, 0xFF, 0x00, 0x0F, 0xF0, 0xF0, 0xF0,
    0xF0, 0x0F, 0x00, 0x0F, 0x0F, 0xF0, 0x0F, 0xF0,
    0xF0, 0xF0, 0xFF, 0x00, 0x0F, 0xF0, 0xF0, 0xF0,
    0xF0, 0x0F, 0x00, 0x0F, 0x0F, 0xF0, 0x0F, 0xF0,
```

```
    0xF0, 0xF0, 0xFF, 0x00, 0x0F, 0xF0, 0xF0, 0xF0,
    0xF0, 0x0F, 0x00, 0x0F, 0x0F, 0xF0, 0x0F, 0xF0
};
```

将自定义模式通知 OpenGL,让它作出准备,一旦可以改变填充模式就进行响应。下述命令可以使 OpenGL 改变填充模式:

```
glEnable(GL_POLYGON_STIPPLE);
```

如果要回到通常的填充模式,只需采用以下命令中止自定义填充模式:

```
glDisable(GL_POLYGON_STIPPLE);
```

DrawPolygonStipple()函数的运行结果如图 4-9 所示。

```
void DrawPolygonStipple()
{
  GLubyte polygonFillPattern[] =
    {
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x03, 0x80, 0x01, 0xC0, 0x06, 0xC0, 0x03, 0x60,
    0x04, 0x60, 0x06, 0x20, 0x04, 0x30, 0x0C, 0x20,
    0x04, 0x18, 0x18, 0x20, 0x04, 0x0C, 0x30, 0x20,
    0x04, 0x06, 0x60, 0x20, 0x44, 0x03, 0xC0, 0x22,
    0x44, 0x01, 0x80, 0x22, 0x44, 0x01, 0x80, 0x22,
    0x44, 0x01, 0x80, 0x22, 0x44, 0x01, 0x80, 0x22,
    0x44, 0x01, 0x80, 0x22, 0x44, 0x01, 0x80, 0x22,
    0x66, 0x01, 0x80, 0x66, 0x33, 0x01, 0x80, 0xCC,
    0x19, 0x81, 0x81, 0x98, 0x0C, 0xC1, 0x83, 0x30,
    0x07, 0xe1, 0x87, 0xe0, 0x03, 0x3f, 0xfc, 0xc0,
    0x03, 0x31, 0x8c, 0xc0, 0x03, 0x33, 0xcc, 0xc0,
    0x06, 0x64, 0x26, 0x60, 0x0c, 0xcc, 0x33, 0x30,
    0x18, 0xcc, 0x33, 0x18, 0x10, 0xc4, 0x23, 0x08,
    0x10, 0x63, 0xC6, 0x08, 0x10, 0x30, 0x0c, 0x08,
    0x10, 0x18, 0x18, 0x08, 0x10, 0x00, 0x00, 0x08
    };

    glPolygonMode(GL_FRONT_AND_BACK,GL_FILL);

    glPolygonStipple(polygonFillPattern);

    glEnable(GL_POLYGON_STIPPLE);

    glBegin(GL_POLYGON);
        glVertex2f(-0.8f,0.8f);
        glVertex2f(-0.8f,-0.80f);
        glVertex2f(0.8f,-0.80f);
```

```
        glVertex2f(0.8f,0.80f);
    glEnd();

    glDisable(GL_POLYGON_STIPPLE);

}
```

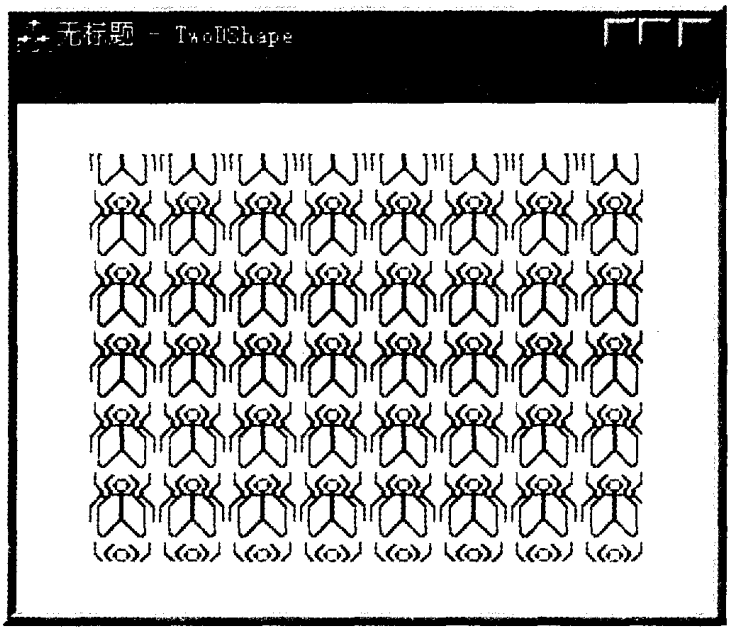

图 4-9　填充多边形

应用多边形填充模式,可以进行半透明处理(透明处理参见第 11 章)和淡入淡出效果处理等。进行淡入淡出效果处理时,最重要的是设置好一系列的多边形填充模式。比如,当进行淡入处理时,代表多边形填充模式的 32 × 32 位位图的位填充应该是由少到多逐渐增加。当然,由于填充位图是一位位图,这种淡入淡出处理的效果不一定很好,但它却是一种较好的思想。

4.4　绘制三角形

4.4.1　绘制三角形

OpenGL 提供了绘制三角形的机制。将 glBegin()函数的参数设为 GL_TRIANGLES 便可完成这项工作。同时,在描述一个三角形时,也应该明确采用什么顺序来排列其顶点。

DrawTriangles()函数画两个三角形,其结果如图 4-10 所示。

```
void DrawTriangles()
{
glPolygonMode(GL_FRONT_AND_BACK,GL_LINE);
```

```
glLineWidth(1.0f);

glBegin(GL_TRIANGLES);
        glVertex2f(-0.5f,0.4f);
        glVertex2f(-0.6f,-0.5f);
        glVertex2f(0.5f,-0.5f);

        glVertex2f(-0.5f,0.8f);
        glVertex2f(-0.5f,0.6f);
        glVertex2f(0.5f,0.5f);
glEnd();

}
```

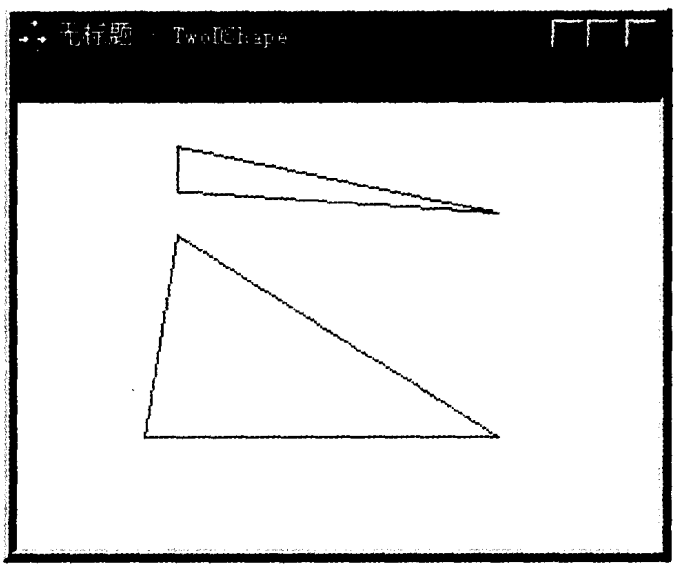

图 4-10　绘制三角形

4.4.2　绘制三角形片

当 glBegin()函数的参数为 GL_TRIANGLE_STRIP 时,可以画一系列的三角形,我们称其为三角形片(Trianle strip)。在 glBegin()/glEnd()函数对中,采用顶点描述三角形时,前三个顶点描述了第一个三角形,第四个顶点可延伸出一个三角形,这个三角形的另外两个顶点是第一个三角形的第二个顶点和第三个顶点。这样,从第四个顶点开始,每增加一个顶点便增加一个三角形。

在定义三角形片中,第一个三角形的顶点的排列顺序是十分重要的,它决定了整个三角形片是前面的还是后面的。当 glFrontFace()函数的参数采用 GL_CCW 时,第一个三角形按逆时针顺序定义时其前面是面向我们的,如图 4-11 示。

在图 4-12 中,第二个三角形的顶点为顶点 2、顶点 4、顶点 3。

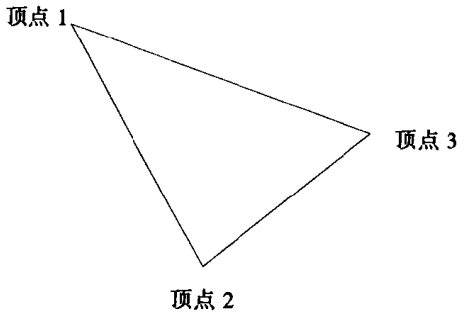

图 4-11　三角形顶点顺序

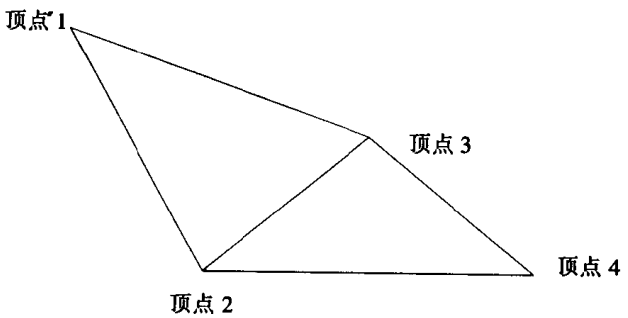

图 4-12　定义三角形片

表 4-1 给出了三角形片中顶点被处理的顺序。

表 4-1　三角形片中顶点处理的顺序

三角形序号	顶点处理顺序
1	顶点 1,顶点 2,顶点 3
2	顶点 3,顶点 2,顶点 4
3	顶点 3,顶点 4,顶点 5
4	顶点 5,顶点 4,顶点 6
5	顶点 5,顶点 6,顶点 7

DrawTriangleStrip()函数绘制一个由 5 个三角形组成的三角形片,其结果如图 4-13 所示。

```
void  DrawTriangleStrip()
{
    glPolygonMode(GL_FRONT_AND_BACK,GL_LINE);
    glLineWidth(1.0f);

    glBegin(GL_TRIANGLE_STRIP);
        glVertex2f(-0.8f,0.3f);
        glVertex2f(-0.8f,-0.3f);
        glVertex2f(-0.3f,0.3f);
```

```
        glVertex2f(0.0f, -0.3f);
        glVertex2f(0.3f,0.5f);
        glVertex2f(0.6f,0.4f);
        glVertex2f(0.8f,0.8f);
    glEnd();
}
```

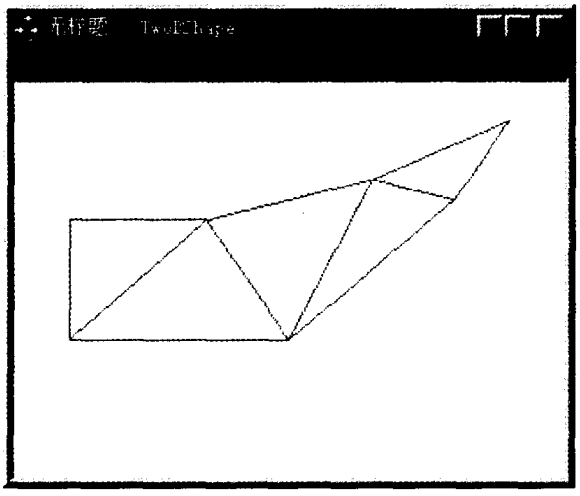

图 4-13　绘制三角形片

4.4.3　绘制三角形扇

在 OpenGL 中,可以用多个三角形构成三角形扇(Triangle fan)。要画三角形扇,必须把 glBegin()函数的参数设为 GL _ TRIANGLE _ FAN。三角形扇由一系列的三角形构成,如图 4-14所示。glBegin()/glEnd()函数对中的前三个顶点定义了第一个三角形,然后每增加一个顶点便增加一个三角形。第二个三角形的顶点依次为顶点 1、顶点 3、顶点 4,第三个三角形的顶点依次为顶点 1、顶点 4、顶点 5,依此类推。表 4-2 说明了三角形扇中顶点被处理的顺序。

表 4-2　三角形扇中顶点处理顺序

三角形序号	顶点处理顺序
1	顶点 1,顶点 2,顶点 3
2	顶点 1,顶点 3,顶点 4
3	顶点 1,顶点 4,顶点 5
4	顶点 1,顶点 5,顶点 6
5	顶点 1,顶点 6,顶点 7

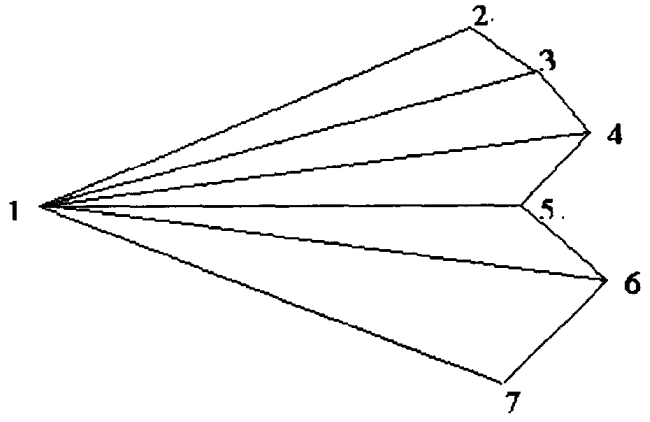

图 4-14　定义三角形扇

DrawTriangleFan()函数绘制一个三角形扇,其结果如图 4-15 所示。

```
void DrawTriangleFan()
{
    glPolygonMode(GL_FRONT_AND_BACK,GL_LINE);
    glLineWidth(1.0f);

    glBegin(GL_TRIANGLE_FAN);
        glVertex2f(-0.5f,-0.3f);
        glVertex2f(0.3f,-0.0f);
        glVertex2f(0.3f,0.3f);
        glVertex2f(0.4f,0.6f);
        glVertex2f(-0.0f,0.6f);
        glVertex2f(-0.3f,0.6f);
    glEnd();
}
```

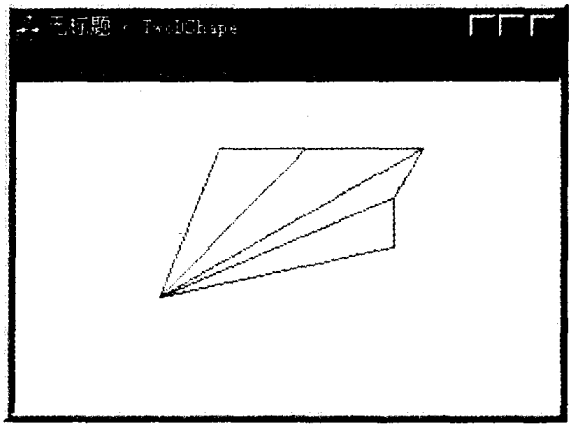

图 4-15　绘制三角形扇

4.5 绘制四边形

4.5.1 绘制四边形

应用 GL_POLYGON 参数,可以画一个四边形(Quadrilateral)。但是,OpenGL 提供同时画一个或一组四边形的机制。将 glBegin()函数的参数设为 GL_QUADS,OpenGL 便可画四边形。绘制四边形与绘制其他图形一样,四边形的顶点的排列顺序也是十分重要的。

DrawQuadrilaterals()函数画一个四边形,其结果如图 4-16 所示。

```
void DrawQuadrilaterals()
{
    glPolygonMode(GL_FRONT_AND_BACK,GL_LINE);
    glLineWidth(1.0f);

    glBegin(GL_QUADS);
        glVertex2f(-0.6f,0.8f);
        glVertex2f(-0.6f,-0.6f);
        glVertex2f(0.5f,0.0f);
        glVertex2f(0.6f,0.5f);
    glEnd();
}
```

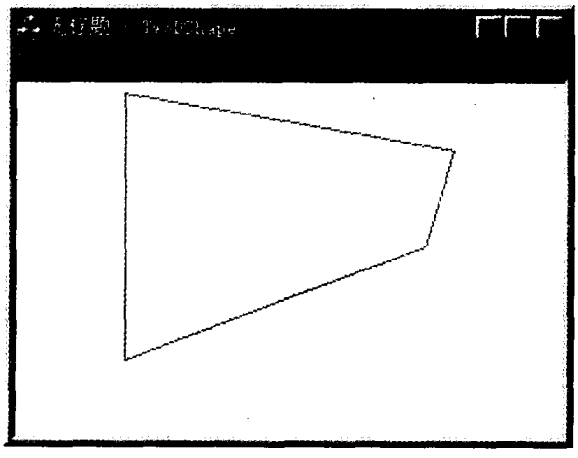

图 4-16 绘制四边形

4.5.2 绘制四边形片

像绘制三角形片一样,四边形也可以进行延伸构成四边形片(Quadrilateral strip),它是一系列四边形连接起来的图形。要绘制四边形片只需将 glBegin()函数的参数设为 GL_QUAD_STRIP 即可。在 glBegin()/glEnd()函数对中,前 4 个顶点决定了第一个四边形,然后每增加

两个顶点便增加一个四边形。当然,四边形片的顶点的顺序也是十分重要的。表 4-3 列出了四边形片中每个四边形的顶点。但实际的顶点标识顺序如图 4-17 所示。

表 4-3 四边形片中的顶点顺序

四边形	四边形顶点
1	顶点 1,顶点 2,顶点 3,顶点 4
2	顶点 3,顶点 4,顶点 5,顶点 6
3	顶点 5,顶点 6,顶点 7,顶点 8
4	顶点 7,顶点 8,顶点 9,顶点 10
5	顶点 9,顶点 10,顶点 11,顶点 12

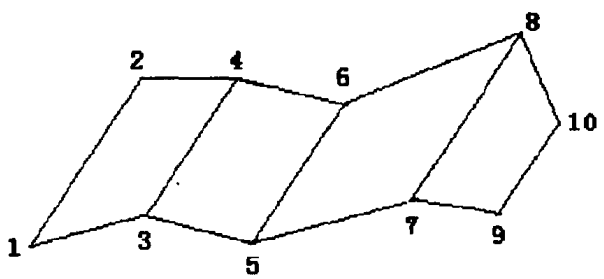

图 4-17 定义四边形片

DrawQuadStrip()函数生成一个四边形片,其结果如图 4-18 所示。

```
void DrawQuadStrip()
{
    glPolygonMode(GL_FRONT_AND_BACK,GL_LINE);
    glLineWidth(1.0f);

    glBegin(GL_QUAD_STRIP);
        glVertex2f(-0.8f,0.3f);
        glVertex2f(-0.7f,0.6f);
        glVertex2f(-0.6f,0.1f);
        glVertex2f(-0.3f,0.6f);
        glVertex2f(-0.1f,-0.2f);
        glVertex2f(0.0f,0.55f);
        glVertex2f(0.3f,-0.4f);
        glVertex2f(0.5f,0.6f);
        glVertex2f(0.8f,-0.4f);
        glVertex2f(0.9f,0.5f);
    glEnd();
}
```

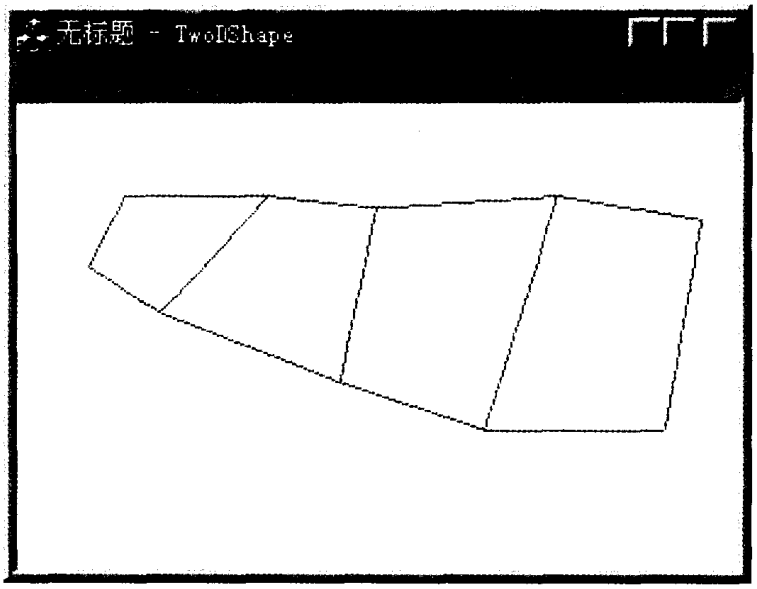

图 4-18　绘制四边形片

4.6　边的可见性

到目前为止,从方法上讲,读者已可以描述并绘制任何复杂的形状。由于物体的拓扑结构是由点线面构成,因此,借助已有的绘制点线面的方法能够绘制出任何复杂的三维形体。从这个角度讲,仅仅面临的问题是如何对物体进行数学描述,如何表示其拓扑结构并形成算法。当然,要生成良好的二维形状和三维形体,其中还有许多问题需要解决。在本章,我们讨论边的可见性。

比如,要绘制如图 4-19 所示的凹多边形,可以通过绘制两个组合的四边形。一个四边形的顶点是(1,2,3,4),另一个四边形的顶点是(3,4,5,6)。但是如果要显示该凹多边形的框架,就会产生一条本不存在的边 3—4,其顶点为第一个四边形的顶点,同时也为第二个四边形的顶点。因此,这条边需要隐藏。

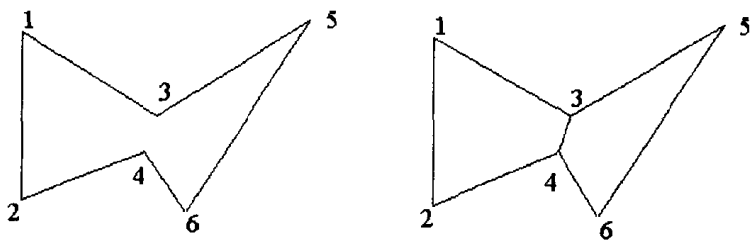

图 4-19　边的可见性

OpenGL 通过 glEdgeFlag()函数来说明边的可见性。该函数位于两个 glVertex()函数之间,说明由这两个顶点构成的边的可见性。如果其参数为 TRUE,则该边是可见的。如果参数为 FALSE,则该边是不可见的。一旦最后定了 glEdgeFlag()函数的参数,此后所绘制的边的

可见性就不会发生改变,除非再一次用不同的参数调用 glEdgeFlag()函数。

DrawNonConvex()函数绘制一个凹多边形,其结果如图 4-20 所示。

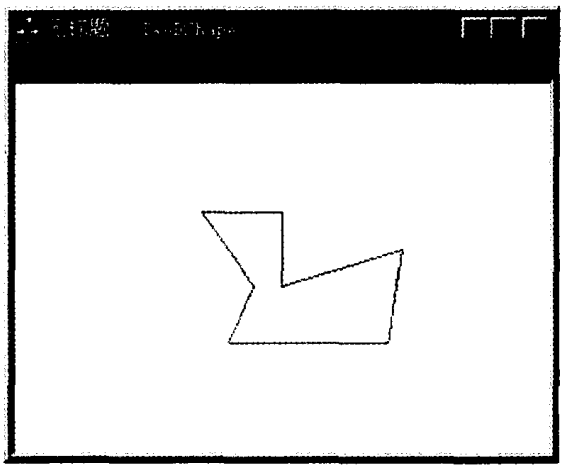

图 4-20　绘制凹多边形

```
void DrawNonConvex()
{
    glPolygonMode(GL_FRONT_AND_BACK,GL_LINE);
    glLineWidth(1.0f);

    glBegin(GL_POLYGON);
        glEdgeFlag(TRUE);
        glVertex2f(-0.3f,0.3f);
        glEdgeFlag(FALSE);
        glVertex2f(-0.1f,-0.1f);
        glEdgeFlag(TRUE);
        glVertex2f(0.0f,-0.1f);
        glVertex2f(0.0f,0.3f);
    glEnd();

    glBegin(GL_POLYGON);
        glEdgeFlag(TRUE);
        glVertex2f(-0.1f,-0.1f);
        glVertex2f(-0.2f,-0.4f);
        glEdgeFlag(FALSE);
        glVertex2f(0.0f,-0.4f);
        glVertex2f(.0f,-0.1f);
    glEnd();

    glBegin(GL_POLYGON);
```

```
        glEdgeFlag(FALSE);
        glVertex2f(0.0f, -0.1f);
        glEdgeFlag(TRUE);
        glVertex2f(0.0f, -0.4f);
        glVertex2f(0.4f, -0.4f);
        glVertex2f(0.45f,0.1f);
    glEnd();
}
```

4.7 由面创建三维形体

一个物体,无论其表面形状多么复杂,计算机都将它视为由若干个小面组成的。通常这些小面采用三角形或四边形。要将一个曲面分割成若干个小面,首先要对曲面进行网格划分,称为有限单元化。其次要获得网格的各个顶点坐标,并对顶点进行数学描述形成单个面,通过面循环构成整个物体的拓扑结构。接着,要赋予顶点信息(例如颜色、纹理、法线等等)。这 3 个重要的步骤通常称为前处理。最后它们均作为输入信息传递给 OpenGL。在这个过程中,网格的划分是极其重要的。小面划分越多,精度就越高。一般地,在物体表面过渡十分急剧的区域需要较多的小面来逼近,在表面较为平坦的区域,离散化的工作就要轻松得多。

例如,要建一个几何中心(形心)在坐标原点的正方体盒子,怎样通过 6 个面来拼接它们?假设有 6 个单位长度的正方形纸板,分别标有“前面”、“后面”、“左面”、“右面”、“底面”、“顶面”等字样,可以按前后左右上下的位置关系把它们拼成一个正方体。在拼接时让标有字的面向外构成盒子的外壳。现在有两种方法使其几何中心位于坐标原点:其一是在拼装时就仔细计算好每个面的位置,使得一拼装好整个盒子的几何中心就位于坐标原点。其二是在任何地方拼装盒子,然后将整个盒子移动,使其几何中心位于坐标原点。这需要执行 OpenGL 变换(参见第 5 章)。

在 OpenGL 中,几乎所有的面都需要自己建立。正如前面已经看到,建立一个面仅仅只需要其顶点就足够了。必须为 OpenGL 提供每个面的顶点,这样 OpenGL 就知道怎样放置面。

OpenGL 有一个缺省的三维网格系统,它是一个两个单位长度的正方体,其几何中心在坐标系的原点,x 轴是自左向右的,y 轴是自下而上的,z 轴是由里向外的。

下面的代码创建一个正方形盒子,并将它绕 y 轴旋转了 70 度,绕 z 轴旋转了 55 度。

```
void DrawCube()
{
    glColor3f(1.0f,0.2f,0.5f);
    glPolygonMode(GL_FRONT_AND_BACK,GL_LINE);
    glLineWidth(1.5f);
    glColor3f(0.2f,0.5f,1.0f);
    glPushMatrix();
        //将盒子绕 y 轴 70 度
        glRotatef(70.0f,0.0f,1.0f,0.0f);

        //将盒子绕 z 轴 55 度
```

```
glRotatef(55.0f,0.0f,0.0f,1.0f);

//前面
glBegin(GL_POLYGON);
        glVertex3f(-0.5f,0.5f,0.5f);
        glVertex3f(-0.5f,-0.5f,0.5f);
        glVertex3f(0.5f,-0.5f,0.5f);
        glVertex3f(0.5f,0.5f,0.5f);
glEnd();

//后面
glBegin(GL_POLYGON);
        glVertex3f(-0.5f,0.5f,-0.5f);
        glVertex3f(0.5f,0.5f,-0.5f);
        glVertex3f(0.5f,-0.5f,-0.5f);
        glVertex3f(-0.5f,-0.5f,-0.5f);
glEnd();

//左面
glBegin(GL_POLYGON);
        glVertex3f(-0.5f,-0.5f,0.5f);
        glVertex3f(-0.5f,0.5f,0.5f);
        glVertex3f(-0.5f,0.5f,-0.5f);
        glVertex3f(-0.5f,-0.5f,-0.5f);
glEnd();

//右面
glBegin(GL_POLYGON);
        glVertex3f(0.5f,-0.5f,0.5f);
        glVertex3f(0.5f,-0.5f,-0.5f);
        glVertex3f(0.5f,0.5f,-0.5f);
        glVertex3f(0.5f,0.5f,0.5f);
glEnd();

//底面
glBegin(GL_POLYGON);
        glVertex3f(-0.5f,-0.5f,0.5f);
        glVertex3f(0.5f,-0.5f,0.5f);
        glVertex3f(0.5f,-0.5f,-0.5f);
        glVertex3f(-0.5f,-0.5f,-0.5f);
glEnd();

//顶面
glBegin(GL_POLYGON);
```

```
            glVertex3f(-0.5f,0.5f,0.5f);
            glVertex3f(0.5f,0.5f,0.5f);
            glVertex3f(0.5f,0.5f,-0.5f);
            glVertex3f(-0.5f,0.5f,-0.5f);
        glEnd();
    glPopMatrix();
}
```

请注意,从外部看,每个面都是前面的。因此,立方体的背面、左面和底面的顶点顺序必须是顺时针的。在用多边形描述一个三维物体时,多边形的前后面区分是十分重要的。如果一个面本来是前面的却当成后面来处理很可能产生不良效果,特别是对其进行光照的时候。

函数 DrawCube()绘制一个立方体,结果如图 4-21 所示。

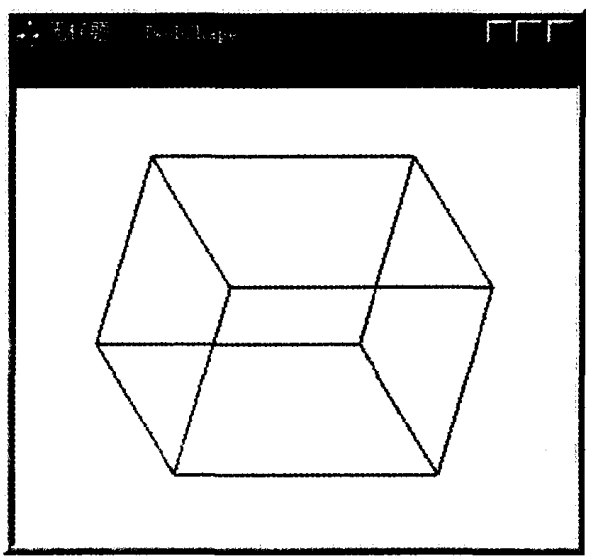

图 4-21　绘制三维物体

从几何的角度讲,用小面来逼近三维物体只需小面的几何顶点就可以了,但是计算机要用这些信息绘制出图形来却有许多问题要解决:第一,顶点、面的法线向量,多边形前后面的界定;第二,面的细分;第三,逼近的精度;第四,精度与开销的矛盾等。

4.8　建立应用程序 TwoDShape

在本节中,我们将把前面介绍的绘制点、直线、多边形等各种二维图形的函数集成在一个应用程序中。其目的主要是希望通过这种方式让读者熟悉本书固定的编程模式。从下一章起,我们将采用一个知识点、一个函数、一次编译连接并运行的模式来逐步集成一个应用程序。

第一步:极小化 OpenGL 及处理界面

(1) 极小化 OpenGL 应用程序

建立一个工程名为 TwoDShape 的工程,并用第 3 章中极小化 OpenGL 应用程序 StepinGl 的步骤极小化 TwoDShape 工程。注意,当添加 DrawScene()函数时,暂时不加入代码。

(2) 添加菜单

添加一个名为 DrawShapes 菜单,即在属性对话框的 Caption 栏内加入 &DrawShapes。增加菜单项,表 4-4 是它们的属性对话框中的 Caption 及 ID 号。

(3) 控制图形

打开 CTwoDShapeView.h,在其首部增加以下语句:

```
enum{Points,    Lines,    LineStipple,    LineStrip,    LineLoop,    Polygons,
     PolygonStipple,    Triangles,    TriangleStrip,    TriangleFan,
     Quads,    QuadStrip,    NonConvex,    Cube};
```

表 4-4

Caption	ID
&Points	IDM_POINTS
&Lines	IDM_LINES
L&ine Stipple	IDM_LINE_STIPPLE
Li&ne Strip	IDM_LINE_STRIP
Lin&e Loop	IDM_LINE_LOOP
Pl&oygon	IDM_POLYGON
Plo&ygon Stipple	IDM_POLYGON_STIPPLE
&Triangles	IDM_TRIANGLES
T&riangle Strip	IDM_TRIANGLE_STRIP
Tri&angle Fan	IDM_TRIANGLE_FAN
&Quadrilaters	IDM_QUADRILATERALS
Q&uad Strip	IDM_QUAD_STRIP
Non Con&vex	IDM_NONCONVEX
Cu&be	IDM_CUBE

为 CTwoDShapeView 类添加一个保护成员变量如下:

```
int        m_twoDShape;
```

其值将是以上枚举类型中的一个,用来控制绘制图形的类型,例如点、直线等。在成员构造函数中初始化:

```
CTwoDShapeView::CTwoDShapeView()
{
    // TODO: add construction code here
    m_pDC = NULL;
    m_twoDShape = Points;
}
```

(4) 响应菜单

确保回到了文件 CTwoDShapeView.cpp 中。为 DrawShapes 菜单的每一个菜单项增加

Windows 消息 COMMAND 及 UPDATE _ COMMAND _ UI。各函数名采用缺省名。

现在,希望通过菜单来选择要画的图形类型。比如,单击 DrawShapes 菜单的 Ponits 项就应该画点。现在,我们来响应诸 COMMAND 及 UPDATE _ COMMAND _ UI 消息:

```
void CTwoDShapeView::OnCube()
{
    // TODO: Add your command handler code here
    m_twoDShape = Cube;
    Invalidate();
}

void CTwoDShapeView::OnUpdateCube(CCmdUI* pCmdUI)
{
    // TODO: Add your command update UI handler code here
    if(m_twoDShape == Cube)
        pCmdUI->SetCheck(TRUE);
    else
        pCmdUI->SetCheck(FALSE);
}

void CTwoDShapeView::OnLineLoop()
{
    // TODO: Add your command handler code here
    m_twoDShape = LineLoop;
    Invalidate();
}

void CTwoDShapeView::OnUpdateLineLoop(CCmdUI* pCmdUI)
{
    // TODO: Add your command update UI handler code here
    if(m_twoDShape == LineLoop)
        pCmdUI->SetCheck(TRUE);
    else
        pCmdUI->SetCheck(FALSE);
}

void CTwoDShapeView::OnLineStipple()
{
    // TODO: Add your command handler code here
    m_twoDShape = LineStipple;
    Invalidate();
}

void CTwoDShapeView::OnUpdateLineStipple(CCmdUI* pCmdUI)
```

```
{
    // TODO: Add your command update UI handler code here
    if(m_twoDShape == LineStipple)
        pCmdUI->SetCheck(TRUE);
    else
        pCmdUI->SetCheck(FALSE);
}

void CTwoDShapeView::OnLineStrip()
{
    // TODO: Add your command handler code here
    m_twoDShape = LineStrip;
    Invalidate();
}

void CTwoDShapeView::OnUpdateLineStrip(CCmdUI* pCmdUI)
{
    // TODO: Add your command update UI handler code here
    if(m_twoDShape == LineStrip)
        pCmdUI->SetCheck(TRUE);
    else
        pCmdUI->SetCheck(FALSE);
}

void CTwoDShapeView::OnLines()
{
    // TODO: Add your command handler code here
    m_twoDShape = Lines;
    Invalidate();
}

void CTwoDShapeView::OnUpdateLines(CCmdUI* pCmdUI)
{
    // TODO: Add your command update UI handler code here
    if(m_twoDShape == Lines)
        pCmdUI->SetCheck(TRUE);
    else
        pCmdUI->SetCheck(FALSE);
}

void CTwoDShapeView::OnNonconvex()
{
    // TODO: Add your command handler code here
```

```
        m_twoDShape = NonConvex;
        Invalidate();
}

void CTwoDShapeView::OnUpdateNonconvex(CCmdUI* pCmdUI)
{
        // TODO: Add your command update UI handler code here
        if(m_twoDShape == NonConvex)
            pCmdUI->SetCheck(TRUE);
        else
            pCmdUI->SetCheck(FALSE);
}

void CTwoDShapeView::OnPoints()
{
        // TODO: Add your command handler code here
        m_twoDShape = Points;
        Invalidate();
}

void CTwoDShapeView::OnUpdatePoints(CCmdUI* pCmdUI)
{
        // TODO: Add your command update UI handler code here
        if(m_twoDShape == Points)
            pCmdUI->SetCheck(TRUE);
        else
            pCmdUI->SetCheck(FALSE);
}

void CTwoDShapeView::OnPolygon()
{
        // TODO: Add your command handler code here
        m_twoDShape = Polygons;
        Invalidate();
}

void CTwoDShapeView::OnUpdatePolygon(CCmdUI* pCmdUI)
{
        // TODO: Add your command update UI handler code here
        if(m_twoDShape == Polygons)
            pCmdUI->SetCheck(TRUE);
        else
            pCmdUI->SetCheck(FALSE);
}
```

```
void CTwoDShapeView::OnPolygonStipple()
{
    // TODO: Add your command handler code here
    m_twoDShape = PolygonStipple;
    Invalidate();
}

void CTwoDShapeView::OnUpdatePolygonStipple(CCmdUI* pCmdUI)
{
    // TODO: Add your command update UI handler code here
    if(m_twoDShape == PolygonStipple)
        pCmdUI->SetCheck(TRUE);
    else
        pCmdUI->SetCheck(FALSE);
}

void CTwoDShapeView::OnQuadStrip()
{
    // TODO: Add your command handler code here
    m_twoDShape = QuadStrip;
    Invalidate();
}

void CTwoDShapeView::OnUpdateQuadStrip(CCmdUI* pCmdUI)
{
    // TODO: Add your command update UI handler code here
    if(m_twoDShape == QuadStrip)
        pCmdUI->SetCheck(TRUE);
    else
        pCmdUI->SetCheck(FALSE);
}

void CTwoDShapeView::OnQuadrilaterals()
{
    // TODO: Add your command handler code here
    m_twoDShape = Quadrilaterals;
    Invalidate();
}

void CTwoDShapeView::OnUpdateQuadrilaterals(CCmdUI* pCmdUI)
{
    // TODO: Add your command update UI handler code here
    if(m_twoDShape == Quadrilaterals)
```

```
        pCmdUI->SetCheck(TRUE);
    else
        pCmdUI->SetCheck(FALSE);
}

void CTwoDShapeView::OnTriangleFan()
{
    // TODO: Add your command handler code here
    m_twoDShape = TriangleFan;
    Invalidate();
}

void CTwoDShapeView::OnUpdateTriangleFan(CCmdUI* pCmdUI)
{
    // TODO: Add your command update UI handler code here
    if(m_twoDShape == TriangleFan)
        pCmdUI->SetCheck(TRUE);
    else
        pCmdUI->SetCheck(FALSE);
}

void CTwoDShapeView::OnTriangleStrip()
{
    // TODO: Add your command handler code here
    m_twoDShape = TriangleStrip;
    Invalidate();
}

void CTwoDShapeView::OnUpdateTriangleStrip(CCmdUI* pCmdUI)
{
    // TODO: Add your command update UI handler code here
    if(m_twoDShape == TriangleStrip)
        pCmdUI->SetCheck(TRUE);
    else
        pCmdUI->SetCheck(FALSE);
}

void CTwoDShapeView::OnTriangles()
{
    // TODO: Add your command handler code here
    m_twoDShape = Triangles;
    Invalidate();
}
```

```
void CTwoDShapeView::OnUpdateTriangles(CCmdUI * pCmdUI)
{
    // TODO: Add your command update UI handler code here
    if(m_twoDShape == Triangles)
        pCmdUI->SetCheck(TRUE);
    else
        pCmdUI->SetCheck(FALSE);
}
```

到目前为止,我们作完了应用程序的用户界面的工作,下面的步骤绘制具体的二维形状。

第二步:绘制二维形状

(1) 为 CTwoDShapeView 类添加保护成员函数

增加以下 14 个保护成员函数,其类型为 void。它们是:

```
void        DrawPoints();
void        DrawLines();
void        DrawLineStipple();
void        DrawLineStrip();
void        DrawLineLoop();
void        DrawPolygons();
void        DrawPolygonStipple();
void        DrawTriangles();
void        DrawTriangleStrip();
void        DrawTriangleFan();
void        DrawQuadrilaterals();
void        DrawQuadStrip();
void        DrawNonConvex();
void        DrawCube();
```

找到本章前几节中的对应函数,并将其代码分别编辑到上述对应的函数中。

(2) 编辑 DrawScene()函数

```
void CTwoDShapeView::DrawScene()
{
    glClearColor(1.0f,1.0f,1.0f,1.0f);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glColor3f(1.0f, 0.0f, 0.0f);
    glShadeModel(GL_FLAT);
    //选择器:
    switch(m_twoDShape)
    {
        case    Points:                 DrawPoints();           break;
        case    Lines:                  DrawLines();            break;
        case    LineStipple:            DrawLineStipple();      break;
```

```
        case    LineStrip:              DrawLineStrip();        break;
        case    LineLoop:               DrawLineLoop();         break;
        case    Polygons:               DrawPolygons();         break;
        case    PolygonStipple:         DrawPolygonStipple();   break;
        case    Triangles:              DrawTriangles();        break;
        case    TriangleStrip:          DrawTriangleStrip();    break;
        case    TriangleFan:            DrawTriangleFan();      break;
        case    Quadrilaterals:         DrawQuadrilaterals();   break;
        case    QuadStrip:              DrawQuadStrip();        break;
        case    NonConvex:              DrawNonConvex();        break;
        case    Cube:                   DrawCube();             break;
    }
    glFinish();
}
```

在函数中,我们用 switch(){... ...}构造了一个选择器,在执行菜单命令时,由该选择器来选择相对应的绘制函数。

(3) 运行程序

编译连接并运行程序。选择菜单中的每一个项,看看结果是否正确。

4.9 TwoDShape 应用程序是如何运行的

从 TwoDShape 应用程序的类 CTwoDShapeView 的文件 TwoDShapeView.cpp 可以看出,TwoDShape应用程序的 OpenGL 初始化工作只是 OpenGL 极小化所必须的工作。程序中的一个重要的变量是 m _ twoDShape。该变量从枚举类型中选择一个值,而这个值是通过响应 DrawShapes 菜单项命令来获得的。在响应菜单命令时,均采用 Invalidate()命令迫使 Windows 重绘用户区。

一系列的 OnUpdate 函数是来标识刚刚对菜单中的某项作了正确的选择,并在菜单项前以一个勾来标识。

程序中,DrawScene()函数是一个重要的函数,用它来绘制具体的二维形状。同时,该函数也是一个绘制不同二维形状的选择器,它根据 m _ twoDShape 的当前值作出绘制图形的选择。该函数的第三条语句把程序绘制的所有图形的颜色均设为红色。

程序的运行结果可能受到窗口高宽比的影响,这时,可在 mainfrm.cpp 文件的 PreCreateWindow()函数中改变以下两条控制窗口大小的语句:

```
BOOL CMainFrame::PreCreateWindow(CREATESTRUCT& cs)
{
    // TODO: Modify the Window class or styles here by modifying
    // the CREATESTRUCT cs/* cs.cx = 190; cs.cy = 220; */
    cs.cx = 300;
    cs.cy = 320;
    return CFrameWnd::PreCreateWindow(cs);
}
```

当 TwoDShape 应用程序在绘制二维形状时，各形状的颜色均为红色，而画立方体盒子时由第三条语句将物体的颜色设为淡蓝色。

当同时去掉 DrawCube()函数中的语句 glPushMatrix();和 glPopMatrix();时，再次运行程序，并以不同的顺序多次绘制盒子，多次绘制二维形状，会有什么结果产生？这一问题留到下一章去解决。

值得非常高兴的是，本书将用到的 Visual C++ 技术基本上就是这些：添加菜单，响应菜单，在 DrawScene()函数中用开关 switch 建立一个选择器，用枚举变量和菜单控制物体的绘制。这是本书所有应用程序的控制模式，这样我们可以把精力集中在 OpenGL 的学习和应用上。

第 5 章　OpenGL 变换

在现实世界中,所有的物体都是三维的。可是这些三维物体又是怎样在计算机中以二维平面的形式来表现的呢? OpenGL 是通过一系列的变换来实现的。这些变换包括几何变换、投影变换、剪切变换、视口变换等。在算法上,它们是通过线性变换和矩阵操作来实现的。

5.1　建立 GLTrans 工程

本章,我们首先建立一个名为 GLTrans 的工程,以便逐步讲解 OpenGL 的各种变换,最后将形成一个 GLTrans 应用程序。请按如下步骤进行:

(1) 极小化工程

建立一个工程名为 GLTrans 的工程,并用第 3 章中极小化 OpenGL 应用程序 StepinGl 的步骤极小化 GLTrans 工程。注意,当添加 DrawScene()函数时,请暂时不加入代码。

(2) 控制变换

打开 CTwoDShapeView.h,在其首部增加以下语句:

```
enum{Cube};
```

添加一个保护成员变量如下:

```
int m_controlTrans;
```

我们将逐步增加这个枚举变量中的内容。

(3) 增加一个求绝对值的函数

在 GLTransView.cpp 的首部增加如下定义:

```
#define ABS(x) ((x)<0?-(x):(x)>0?(x):0)
```

(4) 添加保护成员函数

增加一个保护成员函数 DrawCube( ),该函数绘制一个正方体盒子,我们从这个盒子开始介绍 OpenGL 变换。进行如下编辑:

```
void CGLTransView::DrawCube(GLdouble dLength)
{
    GLdouble dAbsLength;
    dAbsLength = ABS(dLength);

    glPolygonMode(GL_FRONT_AND_BACK,GL_LINE);
    glLineWidth(1.5f);
    glColor3f(0.2f,0.5f,1.0f);

    glBegin(GL_QUAD_STRIP);
```

```
        glVertex3d(-dAbsLength, dAbsLength, dAbsLength);
        glVertex3d(-dAbsLength, -dAbsLength, dAbsLength);
        glVertex3d(dAbsLength, dAbsLength, dAbsLength);
        glVertex3d(dAbsLength, -dAbsLength, dAbsLength);

        glVertex3d(dAbsLength, dAbsLength, -dAbsLength);
        glVertex3d(dAbsLength, -dAbsLength, -dAbsLength);

        glVertex3d(-dAbsLength, dAbsLength, -dAbsLength);
        glVertex3d(-dAbsLength, -dAbsLength, -dAbsLength);

        glVertex3d(-dAbsLength, dAbsLength, dAbsLength);
        glVertex3d(-dAbsLength, -dAbsLength, dAbsLength);
    glEnd();

    glBegin(GL_QUADS);
        glVertex3d(-dAbsLength, dAbsLength, dAbsLength);
        glVertex3d(dAbsLength, dAbsLength, dAbsLength);
        glVertex3d(dAbsLength, dAbsLength, -dAbsLength);
        glVertex3d(-dAbsLength, dAbsLength, -dAbsLength);
    glEnd();

    glBegin(GL_QUADS);
        glVertex3d(-dAbsLength, -dAbsLength, dAbsLength);
        glVertex3d(dAbsLength, -dAbsLength, dAbsLength);
        glVertex3d(dAbsLength, -dAbsLength, -dAbsLength);
        glVertex3d(-dAbsLength, -dAbsLength, -dAbsLength);
    glEnd();
}
```

(5) 编辑 DrawScene()函数

```
void CGLTransView::DrawScene()
{
    glClearColor(1.0f,1.0f,1.0f,1.0f);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glColor3f(1.0f, 0.0f, 0.0f);

    //选择器:
    switch(m_controlTrans)
    {
        case Cube:              DrawCube(0.5);break;
    }
    glFinish();
}
```

以后绘制图形,只需向 switch 语句增加选择即可。这种模式是一个较为简单且通用的结构。

(6) 添加菜单

添加一个 Caption 名为 C&ontrol 的菜单。加入一个菜单项。其 Caption 为 C&ube,其 ID 号为 IDM _ CUBE。

(7) 响应菜单

为 IDM _ CUBE 增加 Windows 消息,并响应如下:

```
void CGLTransView::OnCube()
{
    // TODO: Add your command handler code here
    m_controlTrans = Cube;
    Invalidate();
}

void CGLTransView::OnUpdateCube(CCmdUI* pCmdUI)
{
    // TODO: Add your command update UI handler code here
    if( m_controlTrans == Cube)
        pCmdUI->SetCheck(TRUE);
    else
        pCmdUI->SetCheck(FALSE);
}
```

(8) 运行程序

编译连接并运行程序。单击 Control 菜单的 Cube 项,我们发现 OpenGL 在窗口中绘制了一个正方形,根本就不是一个正方体盒子,如图 5-1。这是为什么? 想一想在第 4 章中,我们将盒子进行了旋转,这正是 OpenGL 变换的作用。试想想,把眼睛(观察点)很近地贴近一个正方体盒子的前面,能看见整个盒子吗?

或者说取景角度不好。当远离盒子观看时,就能看清盒子的全貌了,这相当于观察点远离了场景。

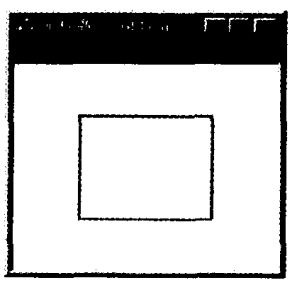

图 5-1 正方体被映射成一个正方形

5.2 投影变换

投影变换就是要将三维物体变换为二维图形。OpenGL 提供了两种投影方式:平行投影(Orthographic Projection)和透视投影(Perspective Projection)。为了使一个盒子看起来像一个盒子,就必须采用透视投影。

在进行投影变换之前应告诉 OpenGL 将操作透视矩阵(Perspective Matrix):

```
glMatrixMode(GL_PROJECTION);
```

透视矩阵是一个 4×4 矩阵。OpenGL 用它作为投影变换的变换矩阵对物体的顶点进行变换,以便能在窗口中显示物体。OpenGL 还采用矩阵堆栈的形式来管理透视矩阵。glMatrixMode()函数采用常量 GL_PROJECTION 在矩阵堆栈中选择透视矩阵。因此透视矩阵必须初始化:

```
glLoadIdentity();
```

glLoadIdentity()用一个 4×4 的单位矩阵初始化当前选定的矩阵。由于当前选定的矩阵类型为透视矩阵,此时该函数就对透视矩阵进行单位化,单位矩阵就成为当前矩阵。OpenGL 总是把每个变换表示成对应的变换矩阵,用它乘以当前矩阵实现变换,并获得新的当前矩阵。当确定采用何种透视方式后,必须告诉 OpenGL 在多大的范围内观察物体,也就是说,必须确定一个视景体(Viewing Volume)。视景体是一个空间区域,一般为长方体或一个台锥。无论场景多么大,位于视景体以外的物体最终都将被剪掉,因而不会在屏幕上显示。构成视景体的 6 个面称为剪切面,它们都是相对观察者而言的。其前面是面向眼睛的,称为近剪切面(Near Plane),其背面称为远剪切面(Far Plane)。

5.2.1 平行投影

平行投影,又叫正交投影或正射投影。它的视景体为一个长方体,如图 6-2 所示。

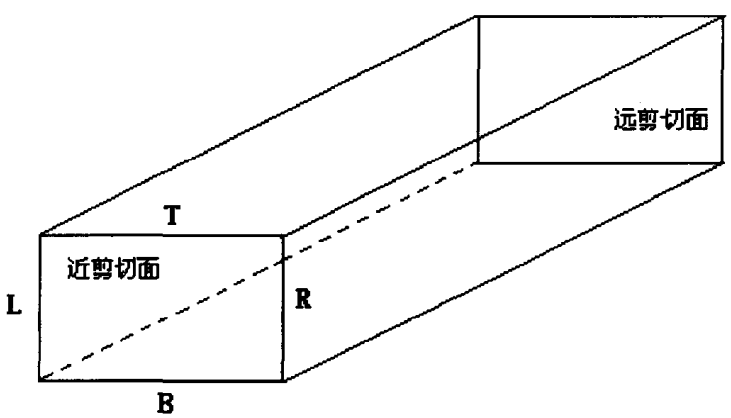

图 5-2 平行视景体

平行投影不会因为物体的远近而改变大小尺寸,这就便于我们观察物体各部分的比例关系。OpenGL 提供两个函数来实现平行投影变换。其一是:

void glOrtho(GLdouble left, GLdouble right, GLdouble bottom, GLdouble top,
GLdouble near, GLdouble far);

该函数建立了一个长方体视景体。其中,left 和 right 为垂直剪切面的坐标;bottom 和 top 为水平剪切面的坐标;near 和 far 分别为近深度剪切面(简称近剪切面)和远深度剪切面(简称远剪切面)。该函数同时确定了一个平行投影变换,近剪切面的左下角点(left,bottom,-near)被映射到窗口的左下角,其右上角点(right,top,-near)被映射到窗口的右上角。-near 确定了近剪切面的位置,-far 则确定了远剪切面的位置。near 和 far 必须具有相同的符号。从算法上讲,该函数创建了一个平行投影变换矩阵,OpenGL 用该矩阵乘以当前透视变换矩阵实现投影变换。

另一个函数是:

void gluOrtho2D(GLdouble left, GLdouble right, GLdouble bottom, GLdouble top);

它是一个 OpenGL 实用库函数。该函数的用途是将二维图像变换到二维平面上。其 near 和 top 的缺省值分别为 -1.0 和 1.0。

5.2.2 透视投影

透视投影使得远离视点(观察点)的物体小,离视点近的物体大。其视景体为一个台锥。如图 5-3 所示。

OpenGL 提供了两个透视投影函数。其一是:

void glFrustum(GLdouble left, GLdouble right, Gldouble bottom, GLdouble top,
GLdouble near, GLdouble far);

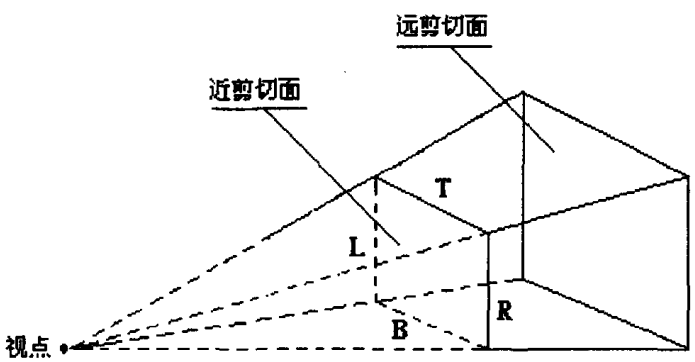

图 5-3 透视视景体

各参数的含义与 glOrtho()函数的参数含义是一致的。但是,near 和 far 表示近剪切面和远剪切面离视点的距离,必须为非负。例如,我们已几次用过以下语句:

glFrustum(-1.0,1.0,-1.0,1.0,0.0,7.0);

类似地,glFrustum()函数建立一个透视变换,同时创建一个透视变换矩阵,用该矩阵乘以当前透视变换矩阵,实现透视变换。

另一个函数是：

void gluPerspective(GLdouble fovy, GLdouble aspect, GLdouble near, GLdouble far);

其中，fovy 定义 y 方向的视角；aspect 说明了在 x 方位的观看区域，即高度与宽度的比率；near 和 far 分别为近剪切面和远剪切面离视点的距离。该函数创建一个对称的视景体，图 5-4 只是一个示意图。

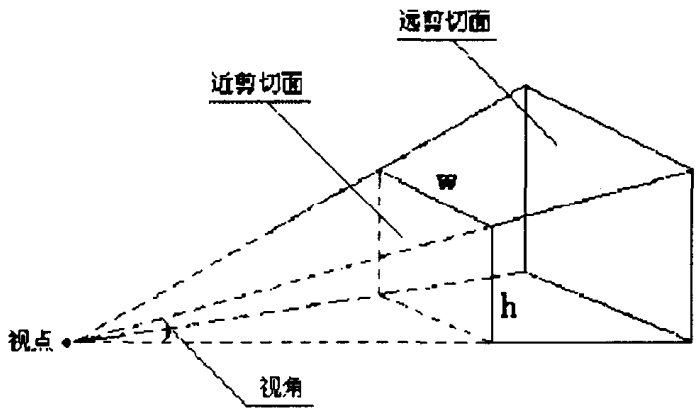

图 5-4　对称视景体

注意，在应用该函数时，视景体的长度与宽度的比率应与视口(Viewport)变换时采用的长度与宽度的比率一致，否则图像会在某方向上拉伸。

在 GLTrans 工程中，我们已建立并启动了透视变换，为什么一个盒子在窗口中仍然只显示成一个正方形？这是因为你的视点正好位于盒子的几何中心，且位于近剪切面上。

最后请注意，当窗口大小改变时，需要重新进行透视(在 OnSize()函数中完成)：

```
void CGLTransView::OnSize(UINT nType, int cx, int cy)
{
    CView::OnSize(nType, cx, cy);

    if(cy > 0)
    {
        if((m_oldRect.right > cx) || (m_oldRect.bottom > cy))
        RedrawWindow();

        m_oldRect.right = cx;
        m_oldRect.bottom = cy;

        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();

        glFrustum(-1.0,1.0,-1.0,1.0,3.0,7.0);
        glViewport(0, 0, cx, cy);
    }
```

5.3 几何变换

5.3.1 一个问题

在讲述几何变换之前，把 CGLTransView::OnSize() 函数作一点改动。将语句

```
glFrustum(-1.0,1.0,-1.0,1.0,0.0,7.0);
```

用

```
glFrustum(-1.0,1.0,-1.0,1.0,3.0,7.0);
```

来替换。此时 OnSize() 函数应为：

```
void CGLTransView::OnSize(UINT nType, int cx, int cy)
{
    CView::OnSize(nType, cx, cy);

    if(cy > 0)
    {
        if((m_oldRect.right > cx) || (m_oldRect.bottom > cy))
          RedrawWindow();

        m_oldRect.right = cx;
        m_oldRect.bottom = cy;

        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();

        glFrustum(-1.0,1.0,-1.0,1.0,3.0,7.0);

        glViewport(0, 0, cx, cy);
    }
}
```

再次运行程序，单击 Control 菜单的 Cube 项，结果窗口上什么也没有了。透视变换已经启动，视景体也已经建立，为什么？这是因为你的盒子位于视景体以外。如图 5-5 所示。

glFrustum(-1.0,1.0,-1.0,1.0,3.0,7.0)设置近剪切面距离视点 3 个单位，但是盒子的几何中心为坐标原点，边长为 1 个单位。可见整个盒子位于视景体之外。

注意，刚才改动了 OnSize()函数的一条语句。本书后面章节若提到“按第 5 章中的方法极小化 OpenGL”则是指在第 3 章所讲的极小化基础上再加上对该函数的改动。

5.3.2 平移变换

解决这一问题的办法就是将盒子沿 z 轴负方向平移使其进入视景体内。将下述代码加入

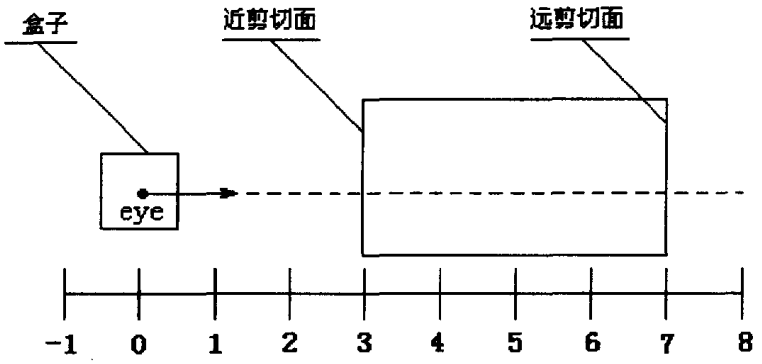

图 5-5　物体的位置与视景体

到 DrawScene()函数中,使它们位于语句 glColor3f(1.0f, 0.0f, 0.0f);之后:

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glTranslatef(0.0f,0.0f,-4.5f);
```

此时,DrawScene()函数变为:

```
void CGLTransView::DrawScene()
{
    glClearColor(1.0f,1.0f,1.0f,1.0f);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glColor3f(1.0f, 0.0f, 0.0f);

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glTranslatef(0.0f,0.0f,-4.5f);

    switch(m_controlTrans)
    {
        case Cube:          DrawCube(0.5);          break;
    }
    glFinish();
}
```

glMatrixMode()函数这次采用 GL_MODELVIEW 参数告诉 OpenGL 要对模型视矩阵(Modelview Matrix)进行操作。glLoadIdentity()函数此时则将模型视矩阵初始化为一个 4×4 的单位阵。glTranslatef()函数实现平移变换。该函数有 3 个参数,分别说明了沿 x、y、z 轴的移动量。事实上,glTranslatef()函数创建一个平移矩阵,用它乘以当前的模型视矩阵实现平移变换,同时得到新的模型视矩阵。语句

```
glTranslatef(0.0f,0.0f,-4.5f);
```

则将盒子沿 z 轴向视点方向移动 4.5 个单位。如图 5-6 所示。

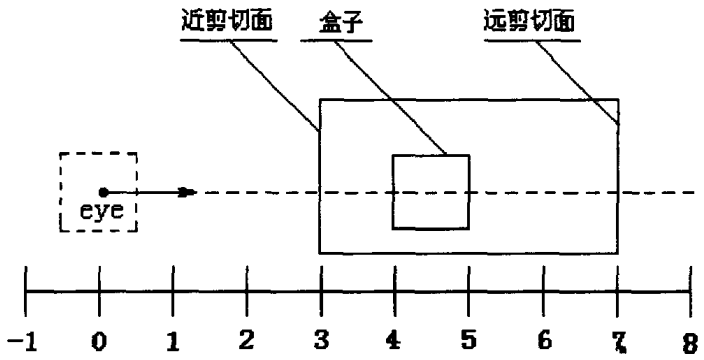

图 5-6　将盒子平移入视景体

重新运行程序。现在，读者可以看见盒子了，如图 5-7 所示。同时，由于这条语句的位置，它会把所有的物体都移动 4.5 个单位，这相当于对视点进行了远离处理。

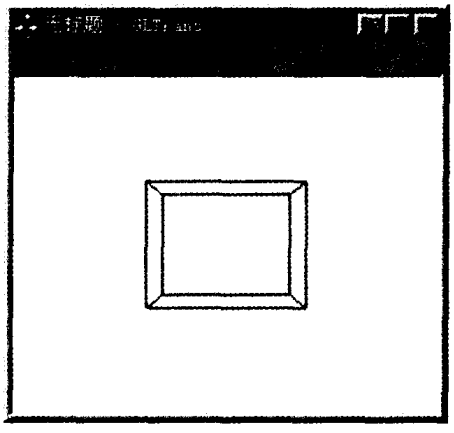

图 5-7　物体远离视点

该显示与你所思考的一致吗？假如把盒子的面以填充的形式输出，比较一下结果的不同之处。

当然，也可以沿其他轴平移物体。比如：

```
glTranslatef(0.2f,0.3f,1.0f);
```

顺便指出，OpenGL 提供了另一个平移函数 gluLookAt()，可以用它观看整个场景。

5.3.3　平移盒子

请按如下步骤完成编程工作：

(1) 编辑 GLTransView.h 文件

在 GLTransView.h 文件的 enum{Cube}中增加：

```
enum{Cube, Translate,Rotate,Scale,Clip,Assemble,Truck};
```

(2) 添加菜单项

在 Control 菜单中新增 6 个菜单项,其 ID 号和 Caption 见表 5-1:

表 5-1

Caption	I D
&Translate	IDM _ TRANSLATE
&Rotate	IDM _ ROTATE
Sc&ale	IDM _ SCALE
C&lip	IDM _ CLIP
Asse&mble	IDM _ ASSEMBLE
Mat&rix Stack	IDM _ MATRIX _ STACK

(3) 响应菜单

响应上述诸消息:

```
void CGLTransView::OnTranslate()
{
    m_controlTrans = Translate;
    Invalidate();
}

void CGLTransView::OnUpdateTranslate(CCmdUI* pCmdUI)
{
    if( m_controlTrans == Translate)
        pCmdUI->SetCheck(TRUE);
    else
        pCmdUI->SetCheck(FALSE);
}

void CGLTransView::OnRotate()
{
    // TODO: Add your command handler code here
    m_controlTrans = Rotate;
    Invalidate();
}

void CGLTransView::OnUpdateRotate(CCmdUI* pCmdUI)
{
    // TODO: Add your command update UI handler code here
    if( m_controlTrans == Rotate)
        pCmdUI->SetCheck(TRUE);
    else
        pCmdUI->SetCheck(FALSE);
```

```
}

void CGLTransView::OnScale()
{
    // TODO: Add your command handler code here
    m_controlTrans = Scale;
    Invalidate();
}

void CGLTransView::OnUpdateScale(CCmdUI* pCmdUI)
{
    // TODO: Add your command update UI handler code here
    if( m_controlTrans == Scale)
        pCmdUI->SetCheck(TRUE);
    else
        pCmdUI->SetCheck(FALSE);
}

void CGLTransView::OnClip()
{
    // TODO: Add your command handler code here
    m_controlTrans = Clip;
    Invalidate();
}

void CGLTransView::OnUpdateClip(CCmdUI* pCmdUI)
{
    // TODO: Add your command update UI handler code here
    if( m_controlTrans == Clip)
        pCmdUI->SetCheck(TRUE);
    else
        pCmdUI->SetCheck(FALSE);
}

void CGLTransView::OnAssemble()
{
    // TODO: Add your command handler code here
    m_controlTrans = Assemble;
    Invalidate();
}

void CGLTransView::OnUpdateAssemble(CCmdUI* pCmdUI)
{
    // TODO: Add your command update UI handler code here
```

```
    if( m_controlTrans == Assemble)
        pCmdUI->SetCheck(TRUE);
    else
        pCmdUI->SetCheck(FALSE);
}

void CGLTransView::OnMatrixStack()
{
    // TODO: Add your command handler code here
    m_controlTrans = Truck;
    Invalidate();
}

void CGLTransView::OnUpdateMatrixStack(CCmdUI * pCmdUI)
{
    // TODO: Add your command update UI handler code here
    if( m_controlTrans == Truck)
        pCmdUI->SetCheck(TRUE);
    else
        pCmdUI->SetCheck(FALSE);
}
```

(4) 添加保护成员函数

增加一个保护成员函数:PfmTranslate(),并编辑如下:

```
void CGLTransView::PfmTranslate()
{
    glPushMatrix();
        glTranslatef(0.2f,0.3f,1.0f);
        DrawCube(0.5);
    glPopMatrix();
}
```

(5) 编辑函数 DrawScene()

在语句 switch(m_controlTrans){... ...}中添加:

```
case Translate:     PfmTranslate();     break;
```

(6) 运行程序

再次编译连接并运行程序。单击 Control 菜单的 Translate,其结果如图 5-8 所示。

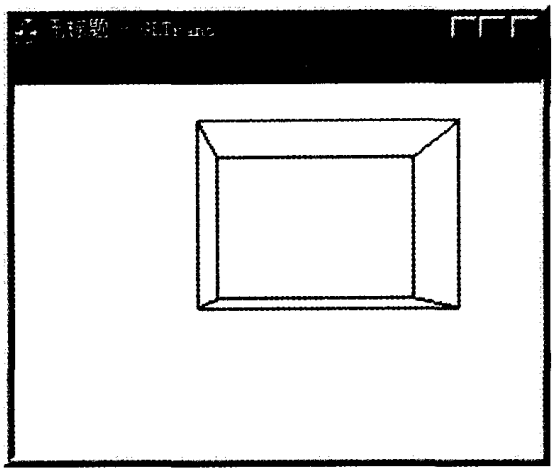

图 5-8　平移盒子

5.3.4　旋转变换

OpenGL 可以对物体进行旋转。例如，要将物体绕 y 轴旋转 60 度可采用如下语句来实现：glRotatef(60.0f,0.0f,1.0f,0.0f);

glRotatef()函数创建一个旋转矩阵，并用它乘以当前模型视矩阵来实现旋转变换。它有 4 个参数，第一个参数指定旋转的度数，后 3 个参数构成一个向量，依次代表该向量的 x、y、z 分量。物体绕该向量旋转时，遵循右手规则。

若绕 x 轴旋转 90 度，则应有：glRotatef(90.0f,1.0f,0.0f,0.0f);

若绕 z 轴旋转 90 度，则应有：glRotatef(90.0f,0.0f,0.0f,1.0f);

注意，角度应以度数的方式来提供。可以绕任意一个给定的向量进行旋转，OpenGL 很方便地提供了这种旋转算法，在第 16 章，我们将基于空间解析几何自己来完成这一算法。请思考如下旋转变换的结果，它对坐标系统有没有影响？

```
glRotatef( - 90.0,1.0f,0.0f,0.0f);
glRotatef( - 90.0,0.0f,0.0f,1.0f);
```

5.3.5　旋转盒子

现在，请按下面的步骤完成编程工作。

(1) 添加保护成员函数

添加一个保护成员函数：PfmRotate()，并编辑如下：

```
void CGLTransView::PfmRotate()
{
    glPushMatrix();
        glRotatef(60.0f,0.0f,1.0f,0.0f);
        glRotated(10.0,1.0,0.0,0.0);
        glRotatef(15.0f,0.0f,0.0f,1.0f);
```

```
        DrawCube(0.5);
    glPopMatrix();
}
```

(2) 编辑函数 DrawScene()

在语句 switch(m _ controlTrans){... ...}中添加:

```
case Translate:        PfmRotate();        break;
```

(3) 运行程序

再次编译连接并运行程序。单击 Control 菜单的 Rotate,其结果如图 5-9:

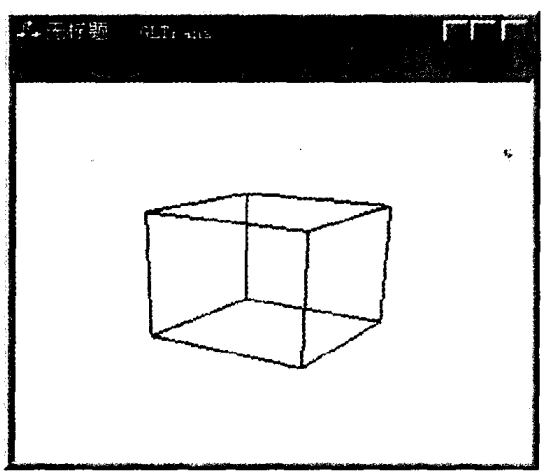

图 5-9 旋转盒子

5.3.6 缩放变换

缩放就是比例放大或缩小。例如,对物体等比放大 1.5 倍可通过如下语句来完成:

```
glScalef(1.5f,1.5f,1.5f);
```

glScalef()函数有 3 个参数,分别表示沿 x、y、z 轴的缩放因子。glScalef()函数创建一个缩放矩阵,OpenGL 用它乘以当前模型视矩阵来完成缩放变换。

当然也可以不进行等比缩放,例如:

```
glScalef(1.0f,3.5f,1.0f);
```

5.3.7 放大盒子

(1) 添加保护成员函数

添加一个保护成员函数:PfmScale(),并编辑如下:

```
void CGLTransView::PfmScale()
```

```
{
    glPushMatrix();
        glScalef(1.5f,1.0f,1.0f);
        DrawCube(0.5);
    glPopMatrix();
}
```

(2) 编辑函数 DrawScene()

在语句 switch(m _ controlTrans){... ...}中添加:

```
case Translate:     PfmScale();     break;
```

(3) 运行程序

再次编译连接并运行程序。单击 Control 菜单的 Scale,其结果如图 5-10:

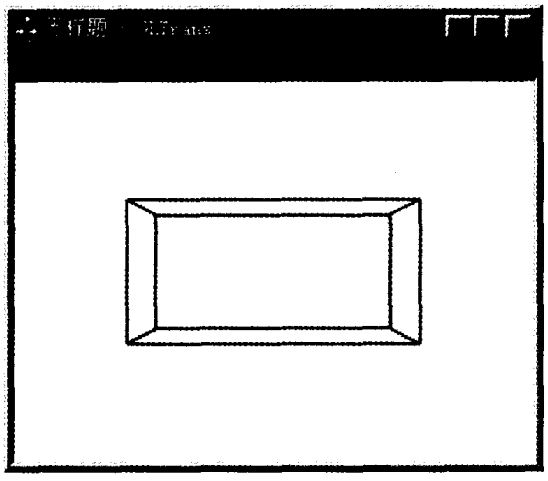

图 5-10 缩放盒子

如果读者用过 3D Studio Max,则知道它有对物体镜像变换的功能,请思考一下 OpenGL 如何实现这一功能?

5.4 视口变换

前面已讨论了 OpenGL 的两个重要的图形学变换:几何变换和透视变换。OpenGL 首先进行的是几何变换,其目的有两个:第一,由于物体间的相对位置关系,物体本身需要进行平移、旋转、缩放变换;第二,将物体放入观察范围内。接着,OpenGL 进行透视变换。透视变换选择并剪切场景,形成视景体,并对其进行透视。要把物体最终显示在窗口上,还需要进行视口变换。视口就是一个用来绘制场景的矩形区域。视口变换决定把场景中的点怎样映射到绘图区。

OpenGL 提供了一个定义视口大小的函数:

```
void glViewPoint(GLint x, GLint y, GLsizei width, GLsizei height );
```

其中，x 和 y 分别说明视口的左下角点在窗口中的位置；width 和 height 分别是视口的宽度和高度，它们都是以像素为单位。视口的缺省区域就是窗口的客户区。

该函数一般在窗口大小改变时就应该进行响应：

```
void CGLTransView::OnSize(UINT nType, int cx, int cy)
{
    CView::OnSize(nType, cx, cy);
    if(cy > 0)
    {
        if((m_oldRect.right > cx) || (m_oldRect.bottom > cy))
            RedrawWindow();
        m_oldRect.right = cx;
        m_oldRect.bottom = cy;
        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        glFrustum(-1.0,1.0,-1.0,1.0,3.0,7.0);
        glViewport(0, 0, cx, cy);
    }
}
```

前面已经指出，视口的宽高比应该和视景体的宽高比一致才使得图形看起来协调。要正确地显示图形，这只是问题的一个方面。一般地，窗口的大小会影响物体的显示，一个正方形在一个非正方形区域内的显示将成为一个长方形。这就是窗口的宽高比的影响。窗口改变的信息是通过 Windows 消息 WM_SIZE 告诉给程序的。它可以获取窗口的大小并实时进行调整。这也是作者认为 OpenGL 极小化中也应该包括对 WM_SIZE 消息响应的原因。

5.5 剪切变换

5.5.1 剪切变换

投影变换为 OpenGL 提供了一个较规则的视景体。但是 OpenGL 可以对剪切面进行操作：

```
void glClipPlane(GLenum plane, const GLdouble * equation );
```

该函数定义一个剪切面。其中，plane 为剪切面序号，其名必须为 GL_CLIP_PLANEi，i 位于 0 与 GL_MAX_CLIP_PLANES - 1 之间。equation 为一个具有 4 个元素的数组，该数组存储一个平面方程 $Ax + By + Cz + d = 0$ 的系数。

视景首先被视景体的剪切面剪切，然后按照 glClipPlane()函数所提供的平面进一步被剪切。剪切的区域满足 $Ax + By + Cz + d < 0$。由于每个剪切面都是凸区域，因而最后形成的视景体也是凸区域。

应该注意，作为一个附加的剪切面，将受到当前几何变换的影响，即当前模型视矩阵的逆矩阵会作为变换矩阵对它进行变换，并把结果存储在观察坐标中。后继的几何变换则对它不

会产生影响。如果一个顶点的观察坐标与一个被变换的平面方程的数组构成的向量的点积非负,则该顶点关于剪切面是内侧的,否则位于剪切面外侧。

在调用附加剪切面时,必须先启动剪切面的功能。要使第 i 个剪切面有效,可以采用:

glEnable(GL_CLIP_PLANEi);

要使其无效则可以采用:

glDisable(GL_CLIP_PLANEi);

这一点是非常有用的,也正体现了 OpenGL 的灵活性。用这个功能可以对整个场景进行剪切,也可以只对一个物体施加剪切变换以制作不同形状的物体。在下面的程序中,将对一个盒子进行剪切。

5.5.2 剪切盒子

按以下步骤剪切盒子:

(1)添加保护成员函数: PfmClip(),并编辑如下:

```
void CGLTransView::PfmClip()
{
    GLdouble dPlaneEquation[4] = {1.0,1.0,1.0,-0.1};
    glPushMatrix();
        glClipPlane(GL_CLIP_PLANE0,dPlaneEquation);
        glEnable(GL_CLIP_PLANE0);
        glRotatef(60.0f,0.0f,1.0f,0.0f);
        glRotated(10.0,1.0,0.0,0.0);
        glRotatef(15.0f,0.0f,0.0f,1.0f);
        DrawCube(0.5);
        glDisable(GL_CLIP_PLANE0);
    glPopMatrix();
}
```

(2) 编辑函数 DrawScene()

在语句 switch(m_controlTrans){... ...}中添加:

```
case Translate:     PfmClip();     break;
```

(3) 运行程序

再次编译连接并运行程序。单击 Control 菜单的 Clip,其结果如图 5-11 所示。

5.6 OpenGL 坐标系

OpenGL 通过执行几何变换、投影变换、剪切变换、视口变换最终生成可视图形。事实上 OpenGL 是将顶点及其他信息进行一系列坐标变换。

在 OpenGL 中,提供给 OpenGL 顶点的坐标称为对象坐标(Object Cordinate),对象坐标采用的是世界坐标(World Coordinate)而不是数学中通用的笛卡尔坐标。模型视矩阵应用于这

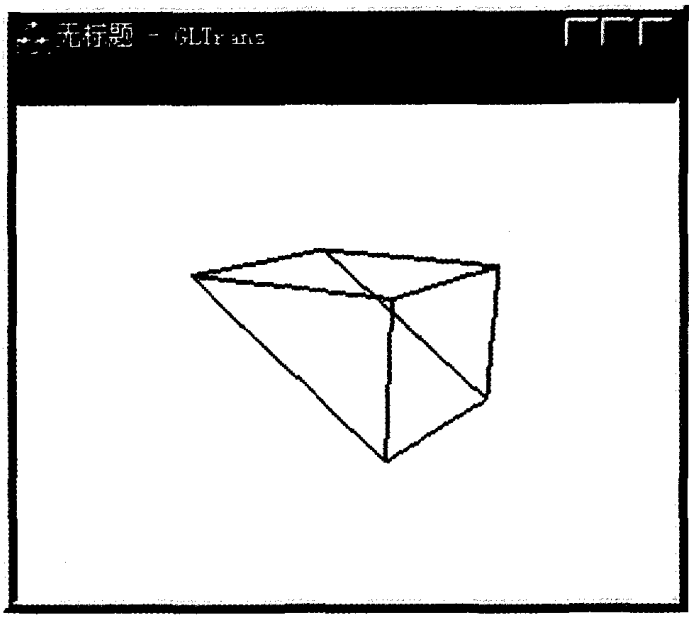

图 5-11　剪切盒子

些坐标生成观察坐标，投影矩阵应用于观察坐标生成剪切坐标，在剪切坐标上执行透视分割产生规格化设备坐标。最后，视口变换则把设备坐标转化成窗口坐标。

这样，OpenGL 涉及到世界坐标系、对象坐标、观察坐标系、剪切坐标系、设备坐标系和窗口坐标系等。设备坐标系和窗口坐标系是 Windows 编程中经常接触到的坐标系。对象坐标、观察坐标、剪切坐标都是四维的。观察坐标的 x、y、z 轴构成左手系，且 x 轴是自左向右的，y 轴是自下而上的，z 轴是由外及里的，好像观察者面对场景一样。透视分割是在剪切坐标中执行的，其变换是将 x、y、z 除以 w。

世界坐标系是一个右手坐标系。在 OpenGL 中，x 轴是自左向右的，y 轴是自下而上的，z 轴是由里及外的。世界坐标系有时也习惯性地称为图形坐标系统。

5.7　集成三维物体

目前，可以简单地将绘制的单个物体显示在窗口中。但是，绘制由多个物体组成的场景，却不是一件容易的事。最重要的问题是如何布置它们。尽管经过精心设计，最后发现有的物体需要再挪动一点，有的需要再倾斜一下，有的则需要再突出一点。当然，可以重新计算它们的空间位置，然后让计算机重新绘制一次。如果按这种方式去工作，绘制 10 个盒子就是一件困难的事。例如，要用本章的 DrawCube()函数绘制一个如图 5-12 放置的盒子，如何完成？DrawCube()函数绘制的盒子总是位于坐标原点，这就需要用 OpenGL 的几何变换来控制各盒子的相对位置。

可以用局部坐标(Local Coordinate)或世界坐标来理解物体的几何变换。世界坐标与笛卡尔坐标没有本质的区别，而局部坐标是相对于物体而言的，其坐标原点往往是物体的形心，并且随着物体的移动而移动。

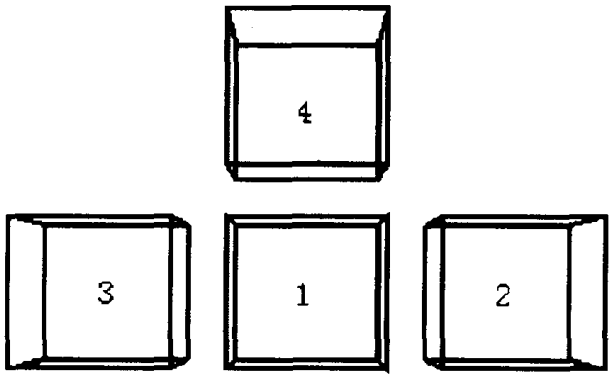

图 5-12　4 个盒子

5.7.1　认识平移

设已用如下两条语句对模型视矩阵进行了初始化：

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
```

接着，执行以下语句：

```
DrawCube(0.3);
```

在调用 DrawCube()函数时，OpenGL 用模型视矩阵乘以盒子的 8 个顶点产生新顶点。由于当前的模型视矩阵是单位阵，故盒子的 8 个顶点仍然是原来的 8 个顶点。此时盒子的局部坐标原点与世界坐标原点是同一个点。添加如下两条语句：

```
glTranslatef(0.8f,0.0f,0.0f);
DrawCube(0.3);
```

绘制了另一个盒子，并不与原盒子重合，而是向右平移了 0.8 个单位。这是因为 OpenGL 执行了平移变换。可以理解为将盒子的局部坐标系向右平移了 0.8 个单位，也可以理解为将世界坐标及其已绘制的物体(除即将绘制的盒子外)向左平移 0.8 个单位。当然，第一种理解要直观些，符合思维习惯，且世界坐标系始终保持不变。如果再执行以下两条语句：

```
glTranslatef(-1.6f,0.0f,0.0f);
DrawCube(0.3);
```

则可以生成第三个盒子，这个盒子距离第二个盒子 1.6 个单位，而离世界坐标原点 0.8 个单位。对世界坐标系而言，这相当于对第三个盒子进行了两次平移变换的合成变换，即向左平移 0.8 个单位，其变换矩阵是两次平移变换的变换矩阵的积。对第二个盒子而言，则相对它进行了一次平移变换，即向左平移了 1.6 个单位。这和线性代数中线性变换是一致的。最后再执行如下两条语句，则产生所要结果。

```
glTranslatef(0.8f,0.8f,0.0f);
DrawCube(0.3);
```

5.7.2 认识旋转和缩放

平移变换较直观,旋转变换却要费解一些,因为旋转变换要改变物体局部坐标系的方位,且一旦改变了坐标系的方位均影响以后物体的绘制。例如,绕 x 轴旋转 90 度后坐标系会发生如图 5-13 所示的变化:

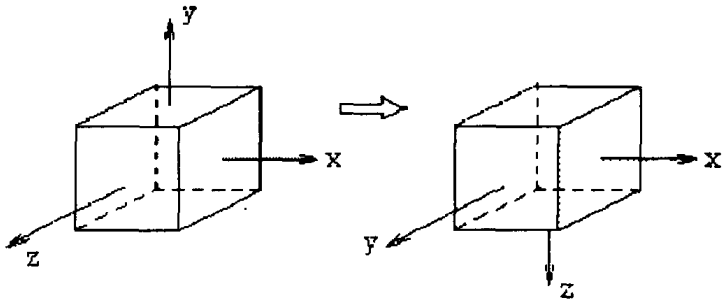

图 5-13 坐标系的旋转

事实上,下述语句将图形坐标系统变换为笛卡尔坐标系统:

```
glRotatef(-90.0,1.0f,0.0f,0.0f);
glRotatef(-90.0,0.0f,0.0f,1.0f);
```

请自学以下代码:

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glTranslatef(0.0f,-0.8f,0.0f);
    DrawCube(0.3);
glTranslatef(0.8f,0.0f,0.0f);
glRotatef(90.0f,0.0f,1.0f,0.0f);
    DrawCube(0.3);
glTranslatef(0.0f,0.0f,-1.6f);
    DrawCube(0.3);
glTranslatef(0.0f,0.8f,0.8f);
    DrawCube(0.3);
```

5.7.3 集成 8 个盒子

请按如下步骤来集成:

(1) 添加保护成员函数

添加一个保护成员函数 PfmAssemble(),并编辑如下:

```
void CGLTransView::PfmAssemble()
{
    glPushMatrix();
        DrawCube(0.3);
```

```
        glTranslatef(0.8f,0.0f,0.0f);
        DrawCube(0.3);
        glTranslatef(-1.6f,0.0f,0.0f);
        DrawCube(0.3);
        glTranslatef(0.8f,0.8f,0.0f);
        DrawCube(0.3);
    glPopMatrix();
    glPushMatrix();
        //向下平移0.8个单位.
        glTranslatef(0.0f,-0.8f,0.0f);
        DrawCube(0.3);
        glTranslatef(0.8f,0.0f,0.0f);
        glRotatef(90.0f,0.0f,1.0f,0.0f);
        DrawCube(0.3);
        glTranslatef(0.0f,0.0f,-1.6f);
        DrawCube(0.3);
        glTranslatef(0.0f,0.8f,0.8f);
        DrawCube(0.3);
    glPopMatrix();
}
```

(2) 编辑函数 DrawScene()

在语句 switch(m_controlTrans){... ...}中添加:

```
case Translate:     PfmAssemble();     break;
```

(3) 运行程序

再次编译连接并运行程序。单击 Control 菜单的 Assemble ,其结果如图 5-14。最后生成 8 个盒子,但在窗口中实际只显示了 7 个盒子,为什么?

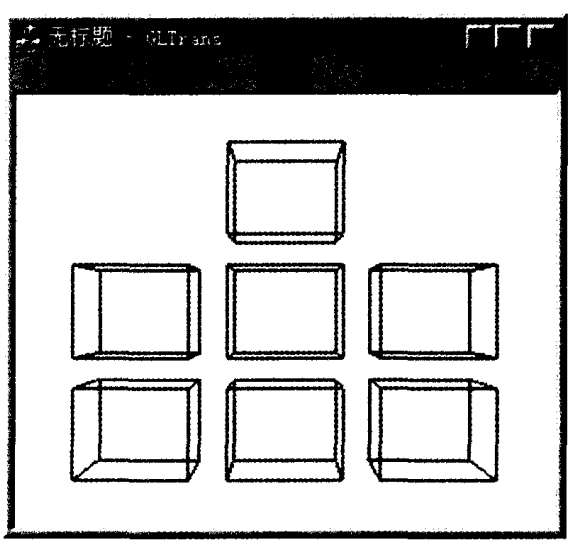

图 5-14 物体间的相关位置

缩放变换只是将坐标系的单位进行缩小或放大，由 glScale()函数的 3 个参数依次说明沿 x,y,z 轴的缩放比例。

在实际的应用中，我们不仅可以用几何变换来组织场景中物体的相关位置，而且还可以利用平移变换、旋转变换和缩放变换的组合来建立自己的取景函数，通过调整取景函数的参数来满足设计和观看场景的需要。

5.8 矩阵堆栈

5.8.1 矩阵堆栈的作用

矩阵堆栈(Matrix Stack)是内存中用来存放矩阵元素的一块区域。其操作原则与普通的堆栈操作原则一样遵循“先进后出，后进先出”。矩阵堆栈为编程提供了一个非常便利的工具，利用它们，可以建立十分复杂的模型，并能清晰地表达模型的运动。我们已经多次运用了矩阵堆栈。对矩阵堆栈的管理是通过如下语句来完成的：

```
glPushMatrix();
glPopMatrix();
```

glPushMatrix()函数将当前矩阵压入栈底，对当前矩阵作一个备份。以后经各种变换产生的当前矩阵被压入堆栈中。当要回到被压入前的位置时，只需用 glPopMatrix()函数弹出矩阵堆栈的顶部矩阵即可。这样，可通过 glPushMatrix()函数与 glPopMatrix()函数的嵌套，灵活地决定自己的位置。例如，要建立一辆有两个货箱的大卡车并模拟其运动，则可通过如下步骤来实现：

```
//    开始
    描述主货箱的运动路径;
    绘制主货箱;

    glPushMatrix();
        描述 4 个主货箱车轮的圆周运动;
        绘制 4 个车轮;
    glPopMatrix();

    glPushMatrix();
        拖箱沿主货箱的运动路径运动,但滞后一段距离;
        绘制拖箱;
        glPushMatrix();
            描述 4 个拖箱车轮的圆周运动;
            绘制 4 个车轮;
        glPopMatrix();
    glPopMatrix();
//    结束
```

上述代码清晰地描述了整个车子的层级关系和运动链接关系。如图 5-15 所示：

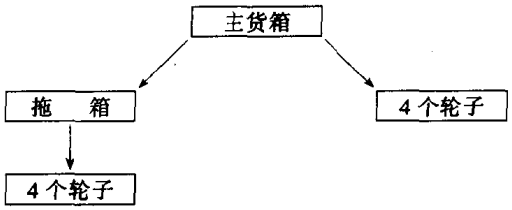

图 5-15　运动链接关系

可见，主货箱的运动必然引起主货箱的 4 个轮子运动。其 4 个轮子还可以在随主货箱运动的同时作自己特有的运动，这种运动就是相对运动，轮子的相对运动则是圆周运动。拖箱总是沿主货箱的运动路径运动，但滞后一段距离。拖箱的运动必然带动 4 个轮子的运动，同时轮子还可以作转动。从上述描述中还可以看出，同级的堆栈间执行的所有变换不相互影响。比如，在画主货箱的 4 个轮子时必然要采用平移变换，可这些平移变换不会影响拖箱的位置。但是，外层堆栈中的所有几何变换一定会影响内层堆栈中物体的位置。即，上级影响下级，但下级不能影响上级。事实上，主货箱和拖箱相当于父子关系，主货箱是父物体，拖箱是子物体。这有点像父亲牵着小孩一样，父亲怎么走，小孩就怎样走，父亲走直路，小孩也走直路，父亲转弯，小孩也跟着转弯，但是，小孩可以在他(她)的父亲周围走来走去，可以走在其父的前面或后面，也可以在父亲的周围转来转去，还可以局部地远离父亲。这种由父物体带动子物体的运动称为正向运动。很明显，利用矩阵堆栈，可以明确地表达物体间的相对位置关系和运动的层级关系。至于怎样利用它们实现反向运动，是一个十分复杂的问题，需要运动学、动力学、数据驱动机制和仿真等等知识，因而这不是本书所能讨论的。

在 OpenGL 中，任何变换类型都有对应的矩阵堆栈，对于几何变换，可用以下语句将模型视矩阵压入堆栈：

```
glMatrixMode(GL_MODELVIEW);
```

对于透视投影变换，则用下述语句将透视矩阵压入堆栈：

```
glMatrixMode(GL_PROJECTION);
```

堆栈中只有一个矩阵时，弹出矩阵操作将会引起错误。最后，矩阵堆栈在保持视点的位置时也有其特别的用处(参见平移变换中的有关内容)。

5.8.2　绘制一辆卡车的平面模型

绘制步骤如下：

(1) 添加保护成员函数

添加 3 个保护成员函数：DrawQuadrangle()，DrawWheel()及 PfmMatrixStack()函数。

DrawQuadrangle()函数绘制一个四边形，它代表车箱。DrawWheel()函数绘制一个圆，它代表车的轮子。PfmMatrixStack()函数则用车箱和轮子连接成整个车子。3 个函数分别编辑如下：

```
void CGLTransView::DrawQuadrangle(GLdouble dQuadWidth,GLdouble dQuadHeight)
{
    GLdouble dWidth,dHeight,dTemp;

    dWidth = ABS(dQuadWidth);
    dHeight = ABS(dQuadHeight);

    if(dWidth < dHeight)
    {
        dTemp = dWidth;
        dWidth = dHeight;
        dHeight = dTemp;
    }

    if(dHeight = = 0.0)
    {
        MessageBox("Error: the Length of Quadrangle is 0.0");
        return;
    }

    dWidth = dWidth/2.0;
    dHeight = dHeight/2.0;

    glPolygonMode(GL_FRONT_AND_BACK,GL_LINE);
    glLineWidth(1.5f);

    glBegin(GL_QUADS);
        glVertex2d(-dWidth,dHeight);
        glVertex2d(-dWidth,-dHeight);
        glVertex2d(dWidth,-dHeight);
        glVertex2d(dWidth,dHeight);
    glEnd();
}

void CGLTransView::DrawWheel(GLdouble dWheelRadius)
{
    GLdouble dRadius = ABS(dWheelRadius);

    GLdouble dStep = 0.1;
    GLdouble dTwoPie = 2.0 * 3.14159265 + dStep;

    glBegin(GL_LINE_STRIP);
        glVertex2d(dRadius,0.0f);
        for(GLdouble dIndex = dStep;dIndex < = dTwoPie;dIndex + = dStep)
```

```
            glVertex2d(dRadius * cos(dIndex),dRadius * sin(dIndex));
        glEnd();
}

void CGLTransView::PfmMatrixStack()
{
        glPolygonMode(GL_FRONT_AND_BACK,GL_LINE);
        glLineWidth(1.5f);

        glColor3f(0.2f,0.5f,1.0f);

        glPushMatrix();
            glTranslatef(0.4f,0.0f,0.0f);

            //主货箱
            DrawQuadrangle(0.4,0.8);

            //连接主货箱与拖箱
            glBegin(GL_LINES);
                glVertex2f(-0.4f,0.0f);
                glVertex2f(-0.7f,0.0f);
            glEnd();

            //主货箱的两个轮子
            glPushMatrix();
                glTranslatef(0.0f,-0.3f,0.0f);
                glTranslatef(-0.3f,0.0f,0.0f);
                DrawWheel(0.1);
                glTranslatef(0.6f,0.0f,0.0f);
                DrawWheel(0.1);
            glPopMatrix();

            glPushMatrix();
                glTranslatef(-1.f,0.0f,0.0f);

                //拖箱
                DrawQuadrangle(0.4,0.6);

                //拖箱的两个轮子
                glPushMatrix();
                    glTranslatef(0.0f,-0.3f,0.0f);
                    glTranslatef(-0.2f,0.0f,0.0f);
                    DrawWheel(0.1);
                    glTranslatef(0.4f,0.0f,0.0f);
```

```
            DrawWheel(0.1);
        glPopMatrix();
    glPopMatrix();
glPopMatrix();
}
```

(2) 编辑函数 DrawScene()

在语句 switch(m _ controlTrans){... ...}中添加:

```
case Translate:    PfmAssemble();    break;
```

(3) 运行程序

再次编译连接并运行程序。单击 Control 菜单的 Matrix Stack ,最后生成一辆车子的平面模型,其结果如图 5-16。

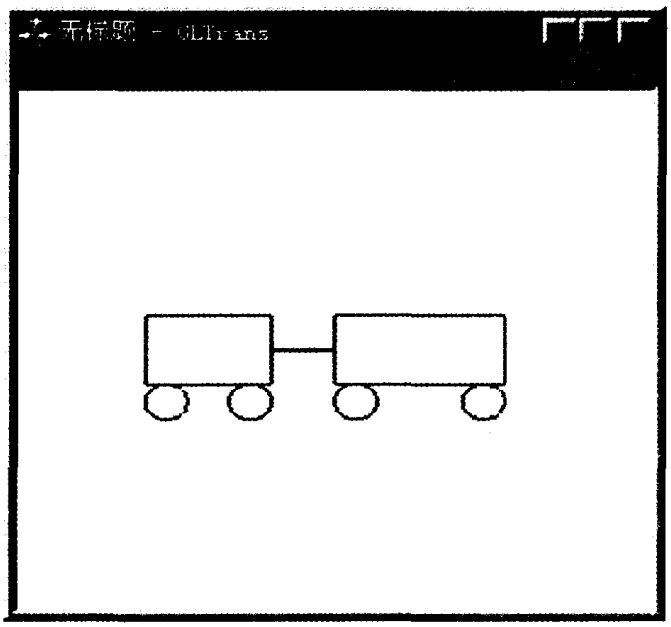

图 5-16 卡车的平面模型

5.9 矩阵操作

本节介绍 OpenGL 中的各种变换的变换矩阵。

(1) 平行投影变换的变换矩阵:

$$\begin{pmatrix} \frac{2}{right-left} & 0 & 0 & t_x \\ 0 & \frac{2}{top-bottom} & 0 & t_y \\ 0 & 0 & \frac{-2}{far-near} & t_x \\ 0 & 0 & 0 & -1 \end{pmatrix}$$

其中 $t_x = -\frac{right + left}{right - left}$, $t_y = -\frac{top + bottom}{top - bottom}$, $t_z = \frac{far + near}{far - near}$

这里,left,right,botom,top,near,far 由 glOrtho(left,right, bottom,top,near,far) ;提供。

(2) 透视投影变换的变换矩阵

$$\begin{pmatrix} \frac{2\cdot near}{right - left} & 0 & A & 0 \\ 0 & \frac{2\cdot near}{top - bottom} & B & 0 \\ 0 & 0 & C & D \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

其中 $A = -\frac{right + left}{right - left}$, $B = \frac{top + bottom}{top - bottom}$, $C = \frac{far + near}{far - near}$, $D = -\frac{2\cdot far\cdot near}{far - near}$

这里,left,right,botom,top,near,far 由 glFrutsum(left,right,bottom, top,near, far); 提供。

(3) 平移变换的变换矩阵

$$\begin{pmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

其中,x、y、z 由 glTranslated(x,y,z);提供。

(4) 缩放变换的变换矩阵

$$\begin{pmatrix} x & 0 & 0 & 0 \\ 0 & y & 0 & 0 \\ 0 & 0 & z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

其中,x、y、z 由 glScaled(x,y,z);提供。

(5) 旋转变换的变换矩阵

旋转变换的变换矩阵要复杂一些。在此,仅给出绕 x 轴、y 轴、z 轴的旋转变换矩阵,绕任意轴的旋转变换的算法可参考第 16 章。

绕 x 轴旋转 a 的变换矩阵:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\alpha) & -\sin(\alpha) & 0 \\ 0 & \sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

绕 y 轴旋转 a 的变换矩阵:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ \cos(\alpha) & 0 & -\sin(\alpha) & 0 \\ 0 & 1 & 0 & 0 \\ \sin(\alpha) & 0 & \cos(\alpha) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

绕 z 轴旋转 a 的变换矩阵:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ \cos(\alpha) & -\sin(\alpha) & 0 & 0 \\ \sin(\alpha) & \cos(\alpha) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

其中，a 由 glRotated()函数的第一个参数来决定。注意角度和弧度的转换。

第6章　显示列表

OpenGL 显示列表(Display List)是一组预先存储起来的 OpenGL 命令。显示列表可以在任何地方被调用,并按顺序立即执行。显示列表是经过编译的,因而执行效率比较高。本章将介绍显示列表的创建、管理及执行。

6.1　创建和执行显示列表

在应用 OpenGL 的命令绘制真彩的三维图形之前,让我们先来介绍 OpenGL 的一个程序运行优化工具:显示列表。OpenGL 显示列表是一系列 OpenGL 命令的高速缓存,不必进行内存管理。当一段代码被大量的调用时,即使是简单的绘图工作,也会降低整体的时间开销,从而提高绘图效率。显示列表驻留于服务器,因而减少了网络传输量,大大地提高了网络性能。

要应用显示列表,首先应创建一个显示列表,由 glNewList()函数和 glEndList()函数来完成。其原型为:

```
void        glNewList( GLuint list, GLenum mode );
void        glEndList( void );
```

其中,list 参数为显示列表序号;mode 参数表示该显示列表被处理的方式,可以为 GL_COMPILE 或 GL_COMPILE_AND_EXECUTE。GL_COMPILE 表示显示列表中的命令仅被编译,GL_COMPILE_AND_EXECUTE 则表示其中的命令不仅被编译而且被立即执行。

下面我们把绘制救生圈(圆环面)的代码放入一个显示列表中:

```
void DrawTrous()
{
    GLint i, j, k;
    GLdouble s, t, x, y, z, twopi;
    Gldouble PI = 3.14159265358979323846;
    //    GLint numc = 64;
    //    GLint numt = 64;
    twopi = 2 * PI;

    //定义一个显示列表
    glNewList(1,GL_COMPILE);
        //第 16 章第 6 节有完整的算法
        for (i = 0; i < 64; i++) {
            glBegin(GL_QUAD_STRIP);
                for (j = 0; j <= 64; j++) {
                    for (k = 1; k >= 0; k--) {
                    s = (i + k) %64 + 0.5;
```

```
                t = j % 64;

                x = (1+0.1 * cos(s * twopi/64)) * cos(t * twopi/64);
                y = (1+0.1 * cos(s * twopi/64)) * sin(t * twopi/64);
                z = 0.1 * sin(s * twopi / 64);
                glVertex3f(x, y, z);
                }
            }
        glEnd();
    }
  glEndList();
}
```

上述代码定义一个显示列表,其序号为 1,用它来绘制一个救生圈。该显示列表用几千个四边形来逼近,所需计算时间是相当多的。若要显示该救生圈的运动,时间耗损必然会更大。但是,如果把它放在显示列表中,已被处理成适合于图形硬件的格式,无论调用多少次,执行时间都极短。顺便指出,第 16 章第 6 节的 my.h 文件中提供了一个完整的绘制圆环面的函数。

执行显示列表的函数是 glCallList()函数。可用下述语句执行上述显示列表:

```
glCallList(1);
```

glCallList()函数执行显示列表,其参数为要被执行的显示列表的序号。尽管显示列表给应用程序提供了一个提高整体性能的机制,但并不能任意使用显示列表。因为启动显示列表也需要一定的开销,如果这种开销不能被执行命令的开销所抵消,那就适得其反了。显示列表可以应用于:

第一,由大量的数据构造的几何模型,且是多次被调用或是几何模型要发生运动。

第二,较高维数的矩阵操作。在图形学算法中,需要涉及大量的矩阵运算,特别是矩阵求逆计算。

第三,光栅位图、图像和纹理操作。应用显示列表可调整像素数据,使其适合硬件处理的格式。

第四,光照计算。光照具有十分复杂的数学运算,如果场景中不同材质的物体较多且需要变化,把材质放入显示列表中,能发挥显示列表的优势。

第五,OpenGL 实用库中标准二次曲面、Bezier 曲线及曲面、NURBS 曲线及曲面的绘制等等。

下面,我们对应用显示列表作一些说明:

第一,显示列表中所用函数的参数值是当前的,不会受今后变化的影响,例如:

```
glNewList(1,GL_COMPILE);
    glBegin(GL_TRIANGLES);
        glColor3f(1.0f, 0.0f, 0.0f);
        glVertex2f(0.0f,0.0f);
        glVertex2f(0.0f,1.0f);
        glVertex2f(1.0f,0.0f);
    glEnd();
```

```
glEndList();
glColor3f(0.0f,1.0f,0.0f);
glBegin(GL_LINES);
    glVertex2f(-0.8f,-0.8f);
    glVertex2f(0.0f,0.0f);
glEnd();
glCallList(1);
```

此时,直线颜色是绿色的,而三角形的颜色将是红色的,不会受重置 glColor3f()函数的影响。又如:

```
GLfloat fPoint1[] = {0.0f,0.0f};
GLfloat fPoint2[] = {0.5f,0.0f};
glNewList(1,GL_COMPILE);
    glBegin(GL_LINES);
        glVertex2fv(fPoint1);
        glVertex2fv(fPoint2);
    glEnd();
glEndList();
fPoint2[0] = 0.5f;
fPoint2[1] = 0.5f;
glBegin(GL_LINES);
    glVertex2fv(fPoint1);
    glVertex2fv(fPoint2);
glEnd();
glCallList(1);
```

位于显示列表中的 glVertex2fv()函数的参数不会受到第二次定义其值的影响,此时该段代码将绘制两条而不是一条直线。

第二,封装在显示列表中的变换操作会对后继绘图产生影响,例如:

```
glNewList(1,GL_COMPILE);
    glBegin(GL_TRIANGLES);
        glVertex2f(0.0f,0.0f);
        glVertex2f(0.0f,0.1f);
        glVertex2f(0.1f,0.0f);
    glEnd();
    glTranslatef(0.1f,0.0f,0.0f);
glEndList();
glCallList(1);
glCallList(1);
glCallList(1);
```

应用上述代码绘制的 3 个三角形是顺序向右排列的,而不是重叠在一起。要使其变换不影响后继绘图,上述代码可改为:

```
glNewList(1,GL_COMPILE);
    glPushMatrix();
      glBegin(GL_TRIANGLES);
            glVertex2f(0.0f,0.0f);
            glVertex2f(0.0f,0.1f);
            glVertex2f(0.1f,0.0f);
        glEnd();
        glTranslatef(0.1f,0.0f,0.0f);
    glPopMatrix();
glEndList();
glCallList(1);
glCallList(1);
glCallList(1);
```

此时,3 个三角形将会重叠在一起。

第三,显示列表不可对其进行外部干预;

第四,显示列表需要精心管理。

但是并非所有的 OpenGL 函数都能在显示列表中获得通过,下面是不能在显示列表中使用的函数:

glIsList(),	glGenLists(),	glDeleteLists(),
glFeedbackBuffer(),	glSelectBuffer(),	glRenderMode(),
glFlush(),	glFinish(),	glIsEnabled(),
glGet() 。		

6.2 管理显示列表

一个显示列表是由其序号来识别的,如果定义一个相同序号的显示列表,则后定义的显示列表会代替已存在的显示列表,这给应用程序带来潜在的危险。为了避免这一危险,可以用 glGenLists()函数来创建一个或多个没有用过的显示列表。其原型为:

```
GLuint  glGenLists( GLsizei range );
```

该函数创建 range 个连续且未被占用的显示列表的序号,返回一个整数。假设返回的整数为 n,则由它创建的 range 个显示列表的序号为 n,n + 1, . . . ,n + range - 1。一旦由该函数分配了多个空的显示列表序号,就可以用它们来创建显示列表而不会发生任何问题。上一节的代码可以优化为:

```
void  DrawTrous()
{
    GLint iListIndex = glGenLists(1);
    if(iListIndex ! = 0)
    {
        glNewList(iListIndex,GL_COMPILE);
            for (i = 0; i < 64; i++) {
```

```
                glBegin(GL_QUAD_STRIP);
                    for (j = 0; j <= 64; j++) {
                        for (k = 1; k >= 0; k--) {
                        s = (i + k) %64 + 0.5;
                        t = j % 64;

                        x = (1+0.1*cos(s*twopi/64))*cos(t*twopi/64);
                        y = (1+0.1*cos(s*twopi/64))*sin(t*twopi/64);
                        z = 0.1 * sin(s * twopi / 64);
                        glVertex3f(x, y, z);
                        }
                    }
                glEnd();
            }
            glEndList();
        }
}
```

在对显示列表进行管理时,还可以对显示列表进行查询。可以给定一个显示列表的序号,查证它是否已被占用。可通过 glIsList ()函数来完成。其原型为:

```
GLboolean glIsList( GLuint list );
```

该函数查证序号为 list 的显示列表是否已经存在,如果存在则返回 GL_TRUE,否则返回 GL_FALSE。

显示列表也可以被删除,可通过 glDeleteLists()函数来完成,其原型为:

```
void glDeleteLists( GLuint list, GLsizei range );
```

该函数删除序号从 list 起的连续 range 个显示列表,同时所有的已给它们分配的地址及内存均被释放。但是这些序号一旦需要还可以再使用。当然,删除零个显示列表是没有意义的。

要直接输出一串位图文字,可以使用显示列表序列来实现。这时需要 glListBase()函数和 glCallLists()函数配合使用。glListBase()函数的原型为:

```
void   glListBase( GLuint base );
```

该函数为 glCallLists()函数设置显示列表序列基数,其中 base 参数可以为任何一个整数,初始值为 0。

glCallLists()函数的原型为:

```
void   glCallLists( GLsizei n, GLenum type, const GLvoid * lists );
```

该函数执行一个显示列表序列。其中,n 参数表示将执行显示列表序列中的个数;type 参数表示 lists 参数值的类型,可以为 GL_BYTE、GL_UNSIGNED_BYTE、GL_SHORT、GL_UNSIGNED_SHORT 、GL_INT、GL_UNSIGNED_INT 、GL_FLOAT、GL_2_BYTES、GL_3_BYTES 和 GL_4_BYTES 。lists 参数则是一个数组地址。其值是一些偏移量,显示列表基数加上偏移量就是将要执行的单个显示列表的序号。

比如,要输出位图文字“A graceful gril !”,可通过如下 3 条语句来完成:

```
wglUseFontBitmaps(wglGetCurrentDC(),0,256,999);
glListBase(999);
glCallLists(16,GL_UNSIGNED_BYTE," A graceful gril!");
```

上述方法自动建立了一个显示列表序列。下面,我们来自己制作一个显示列表序列,并通过 glCallLists()函数一次执行它。

```
GLuint iIdx;
iIdx = glGenLists(4);
glListBase(iIdx);

glNewList(iIdx,GL_COMPILE);
    glBegin(GL_LINES);
        glVertex2f(0.0f,0.0f);
        glVertex2f(-0.1f,0.0f);
    glEnd();
glEndList();

glNewList(iIdx+1,GL_COMPILE);
    glBegin(GL_TRIANGLES);
        glVertex2f(0.0f,0.0f);
        glVertex2f(0.0f,0.1f);
        glVertex2f(0.1f,0.0f);
    glEnd();
glEndList();

glNewList(iIdx+2,GL_COMPILE);
    glBegin(GL_LINES);
        glVertex2f(0.4f,0.0f);
        glVertex2f(0.5f,0.f);
    glEnd();
glEndList();

glNewList(iIdx+3,GL_COMPILE);
    glBegin(GL_LINES);
        glVertex2f(0.6f,0.0f);
        glVertex2f(0.7f,0.f);
    glEnd();
glEndList();

GLbyte BaseArray[4] = {0,1,2,3};
glCallLists(4,GL_BYTE,BaseArray);
```

最后指出,如果位于显示列表内的各函数的参数在应用程序执行 WM_CREATE 消息时

均有确定的值，并且，今后不需要对参数值进行修改，为了让显示列表只创建一次，可以在 OnCreate()函数中进行创建，比如对于 DrawTrous()函数(借用第 5 章中的 OnCreate()函数)，则有：

```
int CGLTransView::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    if (CView::OnCreate(lpCreateStruct) == -1)
        return -1;
    Init();
    DrawTrous();
    return 0;
}
```

第7章　模拟真实世界

前几章，我们所生成的物体均为线框(Wireframe)物体。线框物体能反应物体的结构，不能反应物体的表面细节，有必要对构成物体的各个表面进行填充以形成实体。遗憾的是，用计算机模拟三维真实实体是一个十分复杂的过程。我们所见的物体都是由其材质和光照相互作用的结果。如何生成真实图形是计算机图形学的重要研究内容。真实图形在仿真模拟、几何造型、广告影视、指挥控制、科学计算的可视化等领域有着广泛的应用。

本章将进一步讲述 OpenGL 颜色模型，介绍 OpenGL 光源、光照模型、物体材质、明暗处理、深度测试等生成真实世界的基本方法，最后还要创建一个逻辑调色盘。

7.1　生成真实图形

计算机绘制物体时，必须把三维信息经过某种变换，在二维显示表面(通常是屏幕)上绘制出来。由于投影变换失去深度信息，往往导致图形的二义性。要消除二义性，就必须在绘制时消除实际不可见的线和面，这一过程称为消隐。经过消隐的投影图形才称为物体的真实图形。三维物体在经过消隐后，就应该计算场景中可见面的颜色，以最终确定在显示表面上所要显示的每个像素的颜色。这一过程可概括为建立场景的几何描述、选择场景并进行透视、消隐、真实化。真实化图形是一个十分艰巨的任务，包括颜色模型、明暗效应、光照和材质、纹理映射等计算机图形学的基本内容，涉及数学、物理学、计算机科学、美学、心理学等学科。

感谢 OpenGL，它已为我们作了大量的工作。不需要我们去消隐，不需要我们去设计实现明暗效应、光照和材质、纹理映射的算法。总之，有了 OpenGL，用户就不是制造画笔的工人而是使用画笔的画家。

7.2　OpenGL 的颜色模型

物体的颜色不仅取决于物体本身，还与光照和周围环境光的颜色以及观察者的角度有关。例如，有的物体只反射光，有的物体不仅反射光还会透光。从视角的角度讲，颜色包括 3 个要素：色彩(Hue)、饱和度(Saturation)和亮度(Lightness)。人眼可分辨大约 35 万种颜色，28 种不同的色彩。大多数显示器均采用红、绿、蓝 3 原色，即使用 RGB 颜色模型。RGB 颜色模型是三维直角坐标系统中的一个单位正方体，在正方体的主对角线上，各原色的量相等，产生由暗到亮的白色，即灰度。(0,0,0)为黑色，(1,1,1)则为白色。正方体的其他 6 个角点分别为红、黄、绿、青、蓝和品红。如图 7-1。

OpenGL 可以采用 RGB 颜色模型。红、绿、蓝 3 原色是加性原色，各种颜色能叠加在一起产生复合色，如图 7-2 所示。

OpenGL 通过 glColor()函数来表示物体的颜色，并用颜色缓存(Color buffer)来存储颜色。颜色缓存是由红、绿、蓝、Alpha 位等位平面(Bitplane)组成的，有前缓存(Front buffer)、后缓存(Back Buffer)、左前(Front _ left)和右前(Front _ right)缓存、左后(Back _ left)和右后(Back _

right)缓存。左前缓存是必须的颜色缓存。前缓存是可见缓存,后缓存是不可见缓存。OpenGL 正是采用前后双缓存技术实现动画的。下面介绍对颜色缓存的操作。

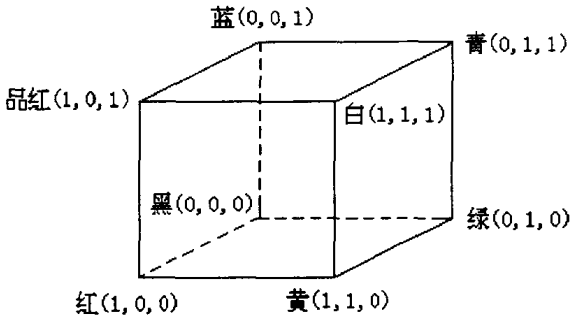

图 7-1 RGB 颜色模型

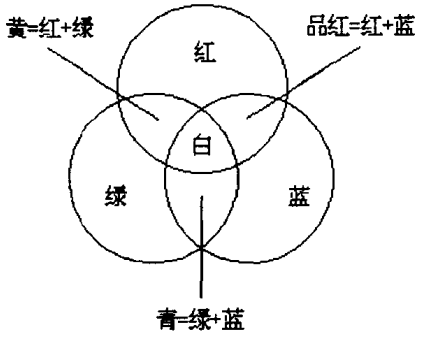

图 7-2 颜色的叠加

(1) 清除颜色缓存:

```
glClear(GL_COLOR_BUFFER_BIT);
```

(2) 设置清除颜色:

```
glClearColor();
```

(3) 屏蔽颜色缓存:

```
glColorMask();
```

该函数有 4 个类型为 GLBoolean 的参数,分别说明红、绿、蓝、Alpha 的可写属性。GL_TRUE 表示允许写该分量,GL_FALSE 表示禁止写该分量。

(4) 选择颜色缓存

当采用双缓存技术时,可以用 glDrawBuffer()函数对颜色缓存进行选择。该函数只能是如下 4 个参数之一:GL_FRONT,GL_BACK,GL_FRONT_AND_BACK 或 GL_NONE。

(5) 交换颜色缓存:

```
swapBuffers();
```

swapBuffers()函数可以交换前后缓存中的颜色,以实现动画。其参数为当前设备上下文

的句柄,下一章我们将介绍其用法。

7.3 建立 Light 工程

现在,建立一个名为 Light 的工程,以便边学边理解,最后将形成一个 Light 应用程序。请完成如下步骤:

(1) 极小化工程

建立一个名为 Light 的工程,并用第 5 章中极小化 OpenGL 应用程序 GLTrans 的步骤极小化 Light 工程。注意,确保 OnSize()函数中对函数 glFrustum()的调用应是:

```
glFrustum(-1.0,1.0,-1.0,1.0,3.0,6.0);
```

(2) 控制物体

打开 CLightView.h,在其首部增加以下语句:

```
enum{Cube,Sphere};
```

添加一个保护成员变量:

```
int   m_selectObject;
```

该变量选择场景中的物体。在本章,我们只绘制 3 个物体:立方体、经平滑处理的立方体和球。重点是 OpenGL 光照。

在构造函数 CLightView()中对其初始化:

```
m_selectObject = Sphere;
```

(3) 增加保护成员函数

增加保护成员函数:DrawCube()和 DrawSphere()。DrawCube()函数绘制一个盒子,并将盒子旋转和放大。DrawSphere()函数绘制一个球,并用显示列表进行管理。分别编辑如下:

```
void CLightView::DrawCube()
{
    glPushMatrix();
        glRotatef(25.0f,1.0f,0.0f,0.0f);
        glRotatef(20.0f,0.0f,1.0f,0.0f);
        glScalef(1.2f,1.2f,1.2f);

        ///////前面
        glBegin(GL_POLYGON);
            glVertex3f(-0.5f,0.5f,0.5f);
            glVertex3f(-0.5f,-0.5f,0.5f);
            glVertex3f(0.5f,-0.5f,0.5f);
            glVertex3f(0.5f,0.5f,0.5f);
        glEnd();

        ///////右面
```

```
        glBegin(GL_POLYGON);
            glVertex3f(0.5f,0.5f,0.5f);
            glVertex3f(0.5f,-0.5f,0.5f);
            glVertex3f(0.5f,-0.5f,-0.5f);
            glVertex3f(0.5f,0.5f,-0.5f);
        glEnd();

        ////////后面
        glBegin(GL_POLYGON);
            glVertex3f(0.5f,0.5f,-0.5f);
            glVertex3f(0.5f,-0.5f,-0.5f);
            glVertex3f(-0.5f,-0.5f,-0.5f);
            glVertex3f(-0.5f,0.5f,-0.5f);
        glEnd();

        //////左面
        glBegin(GL_POLYGON);
            glVertex3f(-0.5f,0.5f,-0.5f);
            glVertex3f(-0.5f,-0.5f,-0.5f);
            glVertex3f(-0.5f,-0.5f,0.5f);
            glVertex3f(-0.5f,0.5f,0.5f);
        glEnd();

        ////////底面
        glBegin(GL_POLYGON);
            glVertex3f(0.5f,-0.5f,0.5f);
            glVertex3f(0.5f,-0.5f,0.5f);
            glVertex3f(0.5f,-0.5f,-0.5f);
            glVertex3f(0.5f,-0.5f,-0.5f);
        glEnd();

        ///////上面
        glBegin(GL_POLYGON);
            glVertex3f(-0.5f,0.5f,-0.5f);
            glVertex3f(-0.5f,0.5f,0.5f);
            glVertex3f(0.5f,0.5f,0.5f);
            glVertex3f(0.5f,0.5f,-0.5f);
        glEnd();
    glPopMatrix();
}

void CLightView::DrawSphere()
{
    GLUquadricObj * quadObj;
```

```
        glNewList(1,GL_COMPILE);
            quadObj = gluNewQuadric();
            gluQuadricDrawStyle(quadObj,GLU_FILL);
            gluQuadricNormals(quadObj,GLU_SMOOTH);
            gluSphere(quadObj,0.7,64,64);
        glEndList();
        glCallList(1);
}
```

在该函数中所用到的实用库函数,将另章介绍。

(4) 添加菜单并响应菜单

添加一个 Caption 名为 S&elect 的菜单。加入两个菜单项。其 Caption 和 ID 号如表 7-1。

表 7-1

Caption	I D
C&ube	IDM_CUBE
Sp&here	IDM_SPHERE

响应如下:

```
void CLightView::OnCube()
{
        // TODO: Add your command handler code here
        m_selectObject = Cube;
        Invalidate();
}

void CLightView::OnUpdateCube(CCmdUI * pCmdUI)
{
        // TODO: Add your command update UI handler code here
        if( m_selectObject == Cube)
            pCmdUI->SetCheck(TRUE);
        else
            pCmdUI->SetCheck(FALSE);
}

void CLightView::OnSphere()
{
        // TODO: Add your command handler code here
        m_selectObject = Sphere;
        Invalidate();
}

void CLightView::OnUpdateSphere(CCmdUI * pCmdUI)
```

```
{
    // TODO: Add your command update UI handler code here
    if( m_selectObject == Sphere)
        pCmdUI->SetCheck(TRUE);
    else
        pCmdUI->SetCheck(FALSE);
}
```

(5) 编辑 DrawScene 函数

```
void CLightView::DrawScene()
{
    glClearColor(1.0f,1.0f,1.0f,1.0f);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glColor3f(0.2f, 0.5f, 0.8f);

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glTranslatef(0.0f,0.0f,-4.5f);

    switch(m_selectObject)
    {
        case Cube:          DrawCube(); break;
        case Sphere:        DrawSphere(); break;
    }
    glFinish();
}
```

(6) 运行程序

编译连接并运行程序,选择 Select 菜单中的两个项,我们发现最后的结果很不理想。难以判断立方体是立方体还是一个平面填充区域,球体是球还是一个被填充的圆。

(7) 增加对话框

增加一个对话框,并按表 7-2 编辑对话框,如图 7-3 所示。

表 7-2

控 件	ID	Caption
对话框	IDD_COLOR__MATERIAL	Color/Matrial
组	IDC_STATIC	Color
组	IDC_STATIC	Color Mask
静态文本框	IDC_STATIC	Red:
静态文本框	IDC_STATIC	Green:
静态文本框	IDC_STATIC	Blue:

（续表）

控　件	ID	Caption
编辑框	IDC_COLOR_RED	
编辑框	IDC_COLOR_GREEN	
编辑框	IDC_COLOR_BLUE	
复选框	IDC_COLOR_MASK_R	Lock Red
复选框	IDC_COLOR_MASK_G	Lock Green
复选框	IDC_COLOR_MASK_B	Lock Blue

为该对话框增加一个类。类名为：CLightMaterial。按表 7-3 添加成员变量。

图 7-3　颜色材质对话框

表 7-3

控 件	成 员	类 型
IDC_COLOR_RED	m_colorRed	float
IDC_COLOR_GREEN	m_colorGreen	float
IDC_COLOR_BLUE	m_colorBlue	float
IDC_COLOR_MASK_R	m_maskRed	BOOL
IDC_COLOR_MASK_G	m_maskGreen	BOOL
IDC_COLOR_MASK_B	m_maskBlue	BOOL

(8) 添加保护成员

添加以下保护成员：

```
GLfloat     m_colorRed;
GLfloat     m_colorGreen;
GLfloat     m_colorBlue;
BOOL        m_bMaskRed;
BOOL        m_bMaskGreen;
BOOL        m_bMaskBlue;
```

在构造函数 CLightView()中初始化上述 6 个成员：

```
m_colorRed = 0.2f;
m_colorGreen = 0.5f;
m_colorBlue = 0.8f;
m_bMaskRed = FALSE;
m_bMaskGreen = FALSE;
m_bMaskBlue = FALSE;
```

m_colorRed,m_colorGreen,m_colorBlue 分别表示颜色的红色、绿色和蓝色分量。m_bMaskRed,m_bMaskGreen,m_bMaskBlue 则分别表示是否对颜色进行屏蔽,如果为 GL_TRUE,则相应分量就不能写入颜色缓存中。

(9) 添加菜单并响应菜单

添加一个 Caption 名为 P&arameter 的菜单。加入一个菜单项,其 Caption 为 C&olor...,其 ID 号为 IDM_COLOR,并响应菜单。注意,在添加消息时更改名字,确保在分别弹出的两个名字输入栏内键入 OnColorMaterial 和 OnUpdateColorMaterial。编辑 OnColorMaterial()如下:

```
void CLightView::OnColorMaterial()
{
    // TODO: Add your command handler code here
    CLightMaterial dlg;

    dlg.m_colorRed = m_colorRed;
    dlg.m_colorGreen = m_colorGreen;
    dlg.m_colorBlue = m_colorBlue;
    dlg.m_maskRed = m_bMaskRed;
    dlg.m_maskGreen = m_bMaskGreen;
    dlg.m_maskBlue = m_bMaskBlue;
    int responeDlg = dlg.DoModal();
    if(responeDlg == IDOK)
    {
        m_colorRed = dlg.m_colorRed;
        m_colorGreen = dlg.m_colorGreen;
        m_colorBlue = dlg.m_colorBlue;
        m_bMaskRed = dlg.m_maskRed;
        m_bMaskGreen = dlg.m_maskGreen;
        m_bMaskBlue = dlg.m_maskBlue;
        Invalidate();
    }
}
```

OnUpdateColorMaterial()函数不作编辑:

```
void CLightView::OnUpdateColorMaterial(CCmdUI * pCmdUI)
{
    // TODO: Add your command update UI handler code here
```

```
}
```

(10) 编辑 DrawScene()函数

在 switch(m_selectObject){... ...}前增加如下语句:

```
glColorMask(!m_bMaskRed,!m_bMaskGreen,!m_bMaskBlue,GL_TRUE);
glColor3f(m_colorRed,m_colorGreen,m_colorBlue);
```

最后,DrawScene()函数如下:

```
void CLightView::DrawScene()
{
    glClearColor(1.0f,1.0f,1.0f,1.0f);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glColor3f(0.2f, 0.5f, 0.8f);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glTranslatef(0.0f,0.0f,-4.5f);

    glColorMask(!m_bMaskRed,!m_bMaskGreen,!m_bMaskBlue,GL_TRUE);
    glColor3f(m_colorRed,m_colorGreen,m_colorBlue);

    switch(m_selectObject)
    {
        case Cube:      DrawCube();      break;
        case Sphere:    DrawSphere();    break;
    }
    glFinish();
}
```

(11) 文件包含

在 LightView.cpp 文件中增加头文件:

```
#include "LightMaterial.h"
```

(12) 运行程序

再次编译连接并运行程序。一开始,程序绘制一个蓝色的球,仍然没有立体感。可以通过 Select 菜单绘制一个立方体,该立方体也没有立体感。用 Parameter 菜单打开 color/material 对话框,修改颜色的各个分量,理解颜色的变化。可以选择对颜色各分量进行屏蔽。当勾选了复选框,则对应分量就不能写入颜色缓存。进行各种组合,并加以理解。

7.4 OpenGL 光照

没有进行光照的三维物体,当其表面按 OpenGL 的方式进行填充时,看起来物体就像失去了深度,没有一点立体感。

7.4.1 光的类型

光照射到物体表面时,可能被物体吸收、反射或透射。光的反射和透射部分进入视觉系统使我们能看见物体。光的颜色由其波长决定。一束白光含有所有可见波长的光。白光照射物体时,只有所有可见光被等量吸收物体才会呈现灰色。如果被不等量吸收,物体会呈现其他的颜色。光的亮度由光强决定。从物体表面反射出来的光的强度取决于光源的位置、光强、物体材质、物体表面位置、物体表面法线和视点的位置。反射光可分为3部分:泛光、漫反射光和镜面反射光。泛光在任何方向上都均匀分布。透过厚厚云层的阳光可视为泛光。泛光可用来模拟从环境中周围物体散射到物体表面再反射出来的光。这种光通常称为环境光。在日常生活中,我们借助于大地、天空、房间的墙壁等反射的光所构成的环境光来看清房间里的物体。任何一种光都能向环境光贡献一定的成份,因而环境光是一种均匀散布的光。漫反射光是物体反射光中向空间均匀反射的光。反射的光强与入射光的入射角的余弦成正比。即是说明,当光线垂直于物体表面时,物体最亮。镜面反射光是朝一定方向反射的光。在光的照射下,光滑物体会形成一个非常亮的区域,这个区域通常称为高光区。物体反射的这部分光称为高光。高光与光源的入射方向、反射方向、视线方向以及物体的反射特性有关。表面光滑的物体,高光区域较小,但亮度很高;而表面粗糙的物体,高光区域大,但亮度小,往往不会引起视觉上的特别注意。一般地,进入人眼的大部分光属于漫反射光。

OpenGL除了可以管理环境光、漫反射光和镜面反射光以外,还可以管理辐射光。一个灯笼里的灯发出的光就可以认为是辐射光。辐射光可以用来模拟自体发光的物体,如灯泡等,以产生特殊效果。

7.4.2 定义光的属性

在OpenGL中,一个光源也是一个对象,它有许多属性。这些属性确定了这个光怎样照射场景中的物体。它们包括环境光的颜色、漫反射光的颜色、镜面反向光的颜色、光源的位置、光源的方向、光的衰减特性等。一种光的颜色也是由红、绿、蓝3原色来决定的。下面的4个数组定义了一种光。

```
GLfloat        m_lightAmb[4] = {0.2f,0.2f,0.2f,1.0f};
GLfloat        m_lightDif[4] = {1.0f,1.0f,1.0f,1.0f};
GLfloat        m_lightSpe[4] = {1.0f,1.0f,1.0f,1.0f};
GLfloat        m_lightPos[4] = {1.0f,1.0f,1.0f,0.0f};
```

由于上述各种光的颜色均相等,因而光的颜色是白色的。前3个数组中的第4个分量表示光的Alpha值。m_lightPos[]数组定义光的位置,其齐次坐标(x,y,z,w)为(1.0f,1.0f,1.0f,0.0f)。如果w=0,OpenGL认为该光源是定向光源,定向光源也叫无限光源。如果w=1.0,OpenGL认为该光源为定位光源,定位光源也叫点光源。定向光源的位置可以认为是无穷远点,因而产生的光是平行光。此时,(x,y,z)不说明光源的位置,而着重说明光源的照射方向,即光的照射方向为(-x,-y,-z)。定位光源是位于场景附近的光源。由于物体的位置不同,定位光源会以不同的角度照射它们。其对象坐标(x,y,z)说明光源的位置。由定位光源可

以创建聚光。

一旦说明了一个光源的属性，就应该让 OpenGL 知道怎样生成光源。生成光源的函数是 glLight()函数。对应地，下述语句定义光的属性：

```
glLightfv(GL_LIGHT0,GL_AMBIENT,m_lightAmb);
glLightfv(GL_LIGHT0,GL_DIFFUSE,m_lightDif);
glLightfv(GL_LIGHT0,GL_SPECULAR,m_lightSpe);
glLightfv(GL_LIGHT0,GL_POSITION,m_lightPos);
```

glLightfv()函数有 3 个参数，第一个参数说明了光源的序号。它只能为 GL_LIGHT0，…，GL_LIGHT7 中的一个。也就是说，OpenGL 至多可以创建 8 盏灯。有图像制作经验的读者一定会知道，光越少效果就会越好。因而这已经足够了。第 3 个参数是一个数组的地址，该数组说明了一个齐次坐标向量。第 2 个参数是一个 OpenGL 常量，只能是表 7-4 中的一个。

表 7-4

常　量	含　义
GL_AMBIENT	设置环境光
GL_DIFFUSE	设置漫反射光
GL_SPECULAR	设置镜面反射光
GL_POSITION	光源位置
GL_SPOT_DIRECTION	聚光照射方向
GL_SPOT_EXPONENT	聚光指数
GL_SPOT_CUTOFF	聚光发散半角
GL_CONSANT_ATTENUATION	常数衰减因子
GL_LINEAR_ATTENUATION	线性衰减因子
GL_QUADRATIC_ATTENUATION	平方衰减因子

OpenGL 总是采用一个开关来控制某种特定功能的有效或无效。对于光照，也必须进行启动：

```
glEnable(GL_LIGHTING);
glEnable(GL_LIGHT0);
```

第一条语句告诉 OpenGL 启动光照，第二条语句则使第一盏灯有效。

7.4.3 光照盒子和球

下面我们给场景添加一个光源，物体的材质暂时采用 OpenGL 缺省材质。

(1) 定义光源属性

为 CLightView 类添加以下保护变量：

```
GLfloat        m_lightAmb[4];
GLfloat        m_lightDif[4];
```

```
GLfloat        m_lightSpe[4];
GLfloat        m_lightPos[4];
```

在构造函数中初始化这些变量：

```
m_lightAmb[0] = 0.2f;          m_lightAmb[1] = 0.2f;
m_lightAmb[2] = 0.2f;          m_lightAmb[3] = 1.0f;

m_lightDif[0] = 1.0f;          m_lightDif[1] = 1.0f;
m_lightDif[2] = 0.3f;          m_lightDif[3] = 1.0f;

m_lightSpe[0] = 1.0f;          m_lightSpe[1] = 1.0f;
m_lightSpe[2] = 1.0f;          m_lightSpe[3] = 1.0f;

m_lightPos[0] = 1.0f;          m_lightPos[1] = 1.0f;
m_lightPos[2] = 1.0f;          m_lightPos[3] = 0.0f;
```

(2) 添加保护成员函数

添加一个保护成员函数 CreateLights(),该函数用来定义并启动光照。编辑如下：

```
void CLightView::CreateLights()
{
    glLightfv(GL_LIGHT0,GL_AMBIENT,m_lightAmb);
    glLightfv(GL_LIGHT0,GL_DIFFUSE,m_lightDif);
    glLightfv(GL_LIGHT0,GL_SPECULAR,m_lightSpe);
    glLightfv(GL_LIGHT0,GL_POSITION,m_lightPos);

    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
}
```

(3) 编辑 DrawScene()函数

在语句 glColor3f(m_colorRed,m_colorGreen,m_colorBlue); 之后增加对 CreateLights()函数的调用：

```
void CLightView::DrawScene()
{
    glClearColor(1.0f,1.0f,1.0f,1.0f);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glColor3f(0.2f, 0.5f, 0.8f);

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glTranslatef(0.0f,0.0f,-4.5f);
    glColorMask(! m_bMaskRed,! m_bMaskGreen,! m_bMaskBlue,GL_TRUE);
    glColor3f(m_colorRed,m_colorGreen,m_colorBlue);
```

```
    CreateLights();

    switch(m_selectObject)
    {
        case Cube:          DrawCube(); break;
        case Sphere:        DrawSphere(); break;
    }
    glFinish();
}
```

(4) 运行程序

再次编译连接并运行程序。现在,一个黄色的球出现了(如图 7-4)! 思考一下光的颜色,理解绘制结果。执行菜单 Select \ Cube,绘制一个立方体。立方体完全是黑色的! OpenGL 光照不正确。这个问题并不是我们没有正确地创建光源,而是 OpenGL 没有获得足够的光照计算条件。其中的一个条件就是物体表面法线。今后,我们会解决这一问题。为什么球就是一个球呢? 绘制球的各函数协调工作,已经考虑了这个问题。

为了对光照有一个深刻的理解,我们借助于对话框对光的变化进行实时观察和模拟。

(5) 增加对话框

增加一个对话框,并按表 7-5 编辑对话框,如图 7-5 所示。

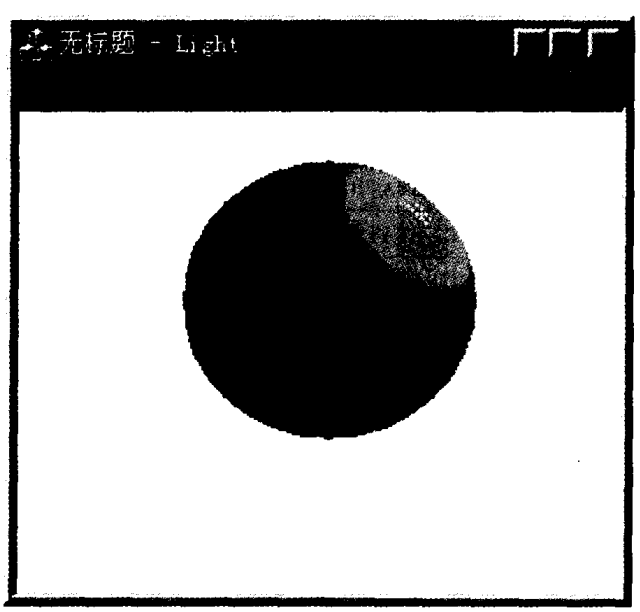

图 7-4 光照球

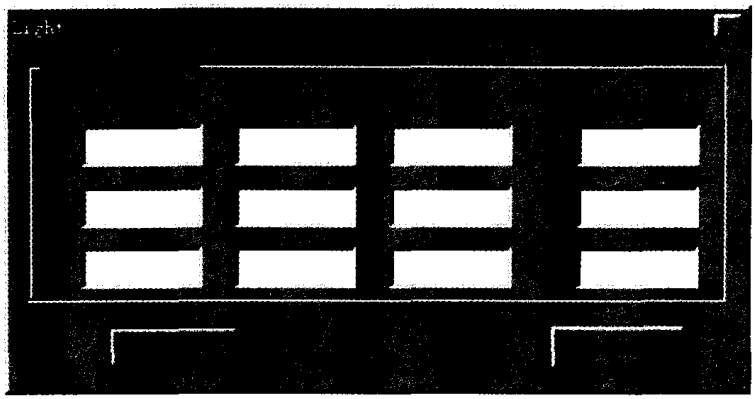

图 7-5　光照对话框

表 7-5

控 件	ID	Caption
对话框	IDD_COLOR__MATERIAL	Light
组	IDC_STATIC	Light0/Directional
静态文本框	IDC_STATIC	R:
静态文本框	IDC_STATIC	G:
静态文本框	IDC_STATIC	B:
静态文本框	IDC_STATIC	Ambient:
静态文本框	IDC_STATIC	Diffuse:
静态文本框	IDC_STATIC	Specular:
静态文本框	IDC_STATIC	X:
静态文本框	IDC_STATIC	Y:
静态文本框	IDC_STATIC	Z:
编辑框	IDC_L_AMBIENT_R	
编辑框	IDC_L_AMBIENT_G	
编辑框	IDC_L_AMBIENT_B	
编辑框	IDC_L_DIFFUSE_R	
编辑框	IDC_L_DIFFUSE_G	
编辑框	IDC_L_DIFFUSE_B	
编辑框	IDC_L_SPECULAR_R	
编辑框	IDC_L_SPECULAR_G	
编辑框	IDC_L_SPECULAR_B	
编辑框	IDC_L_POSITION_X	
编辑框	IDC_L_POSITION_Y	
编辑框	IDC_L_POSITION_Z	

为该对话框增加一个类。类名为:CLighting。按表 7-6 添加成员变量。

表 7-6

控 件	变 量	类 型
IDC_L_AMBIENT_R	m_lightAmbientR	float
IDC_L_AMBIENT_G	m_lightAmbientG	float
IDC_L_AMBIENT_B	m_lightAmbientB	float
IDC_L_DIFFUSE_R	m_lightDiffuseR	float
IDC_L_DIFFUSE_G	m_lightDiffuseG	float
IDC_L_DIFFUSE_B	m_lightDiffuseB	float
IDC_L_SPECULAR_R	m_lightSpecularR	float
IDC_L_SPECULAR_G	m_lightSpecularG	float
IDC_L_SPECULAR_B	m_lightSpecularB	float
IDC_L_POSITION_X	m_lightPositonX	float
IDC_L_POSITION_Y	m_lightPositonY	float
IDC_L_POSITION_Z	m_lightPositonZ	float

(6)添加菜单项

在 Parameter 菜单中加入一个菜单项。其 Caption 为 &Light...,其 ID 号为 IDM_LIGHT。并响应菜单。此时对 OnUpdateLight()函数不作任何编辑工作,OnLight()函数编辑如下:

```
void CLightView::OnLight()
{
    // TODO: Add your command handler code here
    CLighting lightDlg;

    lightDlg.m_lightAmbientR = m_lightAmb[0];
    lightDlg.m_lightAmbientG = m_lightAmb[1];
    lightDlg.m_lightAmbientB = m_lightAmb[2];

    lightDlg.m_lightDiffuseR = m_lightDif[0];
    lightDlg.m_lightDiffuseG = m_lightDif[1];
    lightDlg.m_lightDiffuseB = m_lightDif[2];

    lightDlg.m_lightSpecularR = m_lightSpe[0];
    lightDlg.m_lightSpecularG = m_lightSpe[1];
    lightDlg.m_lightSpecularB = m_lightSpe[2];

    lightDlg.m_lightPositionX = m_lightPos[0];
    lightDlg.m_lightPositionY = m_lightPos[1];
    lightDlg.m_lightPositionZ = m_lightPos[2];

    int responeDlg = lightDlg.DoModal();
```

```
    if(responeDlg == IDOK)
    {
        m_lightAmb[0] = lightDlg.m_lightAmbientR;
        m_lightAmb[1] = lightDlg.m_lightAmbientG;
        m_lightAmb[2] = lightDlg.m_lightAmbientB;

        m_lightDif[0] = lightDlg.m_lightDiffuseR;
        m_lightDif[1] = lightDlg.m_lightDiffuseG;
        m_lightDif[2] = lightDlg.m_lightDiffuseB;

        m_lightSpe[0] = lightDlg.m_lightSpecularR;
        m_lightSpe[1] = lightDlg.m_lightSpecularG;
        m_lightSpe[2] = lightDlg.m_lightSpecularB;

        m_lightPos[0] = lightDlg.m_lightPositionX;
        m_lightPos[1] = lightDlg.m_lightPositionY;
        m_lightPos[2] = lightDlg.m_lightPositionZ;

        Invalidate();
    }
}
void CLightView::OnUpdateLight(CCmdUI * pCmdUI)
{
    // TODO: Add your command update UI handler code here

}
```

(7) 文件包含

在 LightView.cpp 文件中增加头文件:

```
#include "Lighting.h"
```

(8) 运行程序

再次编译连接并运行程序。执行菜单 Parameter \ Light...,打开光照对话框,改变光的各种属性,看看有什么变化。通过环境光、漫反射光、镜面反射光的颜色的变化和定向光源方向的改变,实时反映了 OpenGL 对光照的模拟。可以预见,无论作什么变化,立方体总是一个没有进行正确光照的物体。它始终呈一个黑体!

7.5 物体的材质

材质是指物体表面对光的反射特性。一种材质由环境色、扩散色、镜面高亮色等组成,分别说明了它对环境光、漫反射光和镜面反射光反射的多少,即反射率。材质影响物体的颜色、反光度、透明度等。假设一个只反射蓝光的物体,在一束白光的照射下就呈蓝色。一般来说,

物体对环境光和漫反射光的反射方式是相同的，因而对环境光和漫反射光的反射率基本上是相等的。但是，物体对于镜面反射光的反射率是由镜面反射光决定的，且物体的高亮色基本上与镜面反射光的颜色是相同的。如果一个物体在青色光的照射下呈蓝色，只需设定物体的环境色和扩散色均为蓝色即可。为了让它看起来有高光效果，可设定其镜面高亮色为青色。在编程和图像制作的过程中，有时，我们不去遵循这些物理规律，而是任意地利用它们的组合来产生一些特殊的效果。

7.5.1 定义材质

要定义材质，首先要说明物体的材质属性。这些属性确定物体对环境光、漫反射光、镜面反射光的反射率及光度。可做如下定义：

```
GLfloat    m_materialAmb[4] = {0.2f,0.2f,0.2f,1.0f};
GLfloat    m_materialDif[4] = {0.8f,0.0f,0.0f,1.0f};
GLfloat    m_materialSpe[4] = {1.0f,1.0f,1.0f,1.0f};
GLfloat    m_matShininess = 10.0f;
```

每个数组的前3个元素依次说明了物体对红光、绿光、蓝光的反射率。比如 m_materialDif[ ]数组定义了该种材质只反射红光，而不反射绿光和蓝光。第4个元素反应材质的透明度。m_matShininess 则定义物体的光度。

在定义了材质的基本属性之后，也应该把这些属性通知给 OpenGL：

```
glMaterialfv(GL_FRONT,GL_AMBIENT,m_materialAmb);
glMaterialfv(GL_FRONT,GL_DIFFUSE,m_materialDif);
glMaterialfv(GL_FRONT,GL_SPECULAR,m_materialSpe);
glMaterialf(GL_FRONT,GL_SHININESS,m_matShininess);
```

glMaterial()函数有3个参数。第一个参数指定要定义的多边形的面。有3个可选常量：GL_FRONT、GL_BACK 和 GL_FRONT_AND_BACK，它们分别说明了多边形的前面、后面或双面。区分一个多边形的前后面是十分重要的，因为后面往往是不可见的。第2个参数也是一个 OpenGL 常量，通过它告诉 OpenGL 定义材质的哪一个属性。其可选常量如表7-7所示。

表 7-7

常　量	含　义	缺省值
GL_AMBIENT	设置环境色	(0.2,0.2,0.2,1.0)
GL_DIFFUSE	设置扩散色	(0.8,0.8,0.8,1.0)
GL_AMBIENT_AND_DIFFUSE	设置环境色和扩散色	
GL_SPECULAR	设置镜面高亮色	(0.0,0.0,0.0,1.0)
GL_EMISSION	设置辐射光色	(0.0,0.0,0.0,1.0)
GL_SHININESS	设置光亮度	0
GL_COLOR_INDEXES	用于颜色索引模式	

第 4 条语句的第 3 个参数定义了物体的光度，决定镜面反射光中多少光被反射，其取值范围为[0.0,128.0]。数值越小，反射越强，物体高光越明显。

注意，一旦定义了材质，则以后绘制的物体就采用这种材质，除非重新生成材质或采用矩阵堆栈进行管理。

7.5.2 辐射光

辐射光是图像处理技术中用来模拟发光体的一种光，比如用来模拟灯泡等自体发光的物体。在 OpenGL 中，自体发光被认为是物体的某种属性。可类似定义辐射光：

```
GLfloat   m_materialEmi[4] = {0.4f,0.4f,0.4f,1.0f};
glMaterialfv(GL_FRONT,GL_EMISSION,m_materialEmi);
```

7.5.3 材质和光的相互作用

一个光源决定了光的颜色、反射特性、强度和入射光的方向。物体的几何性质及材质决定反射光的颜色、反射特性、强度。光和物体间的相互作用远比我们能够实时模拟的情况复杂。比如，在不同光照条件下模拟人的皮肤的颜色是一件十分困难的事。OpenGL 在逼真显示和实时绘制之间通过以下方法实现均衡：只在几何顶点上进行光照运算，对在所绘制的每个像素上执行光照运算的复杂处理进行简化。

光和材质的颜色都采用红、绿、蓝 3 原色来合成。例如，有一个黄色光源(0.8,0.8,0.0)，一个对环境光、漫反射光的红、绿、蓝 3 原色的反射率均为 0.5 的物体，则进入视觉系统的颜色为：R = 0.8 × 0.5 = 0.4，G = 0.8 × 0.5 = 0.4，B = 0.0 × 0.5 = 0.0。最后复合成一个饱和度更低的黄色物体。在多个光源的照射下，物体的颜色将进行线性叠加。

7.5.4 制作材质

现在，我们来制作物体的材质，并采用对话框的形式来实时模拟材质的变化。

(1) 定义材质属性

为 CLightView 类添加以下保护变量：

```
GLfloat          m_materialAmb[4];
GLfloat          m_materialDif[4];
GLfloat          m_materialSpe[4];
GLfloat          m_materialEmi[4];
GLfloat          m_matshininess;
BOOL             m_bMaterial;
```

在构造函数中初始化这些变量：

```
m_materialAmb[0] = 0.2f;          m_materialAmb[1] = 0.2f;
m_materialAmb[2] = 0.2f;          m_materialAmb[3] = 1.0f;
```

```
m_materialDif[0] = 0.8f;          m_materialDif[1] = 0.8f;
m_materialDif[2] = 0.8f;          m_materialDif[3] = 1.0f;

m_materialSpe[0] = 0.0f;          m_materialSpe[1] = 0.0f;
m_materialSpe[2] = 0.0f;          m_materialSpe[3] = 1.0f;

m_materialEmi[0] = 0.0f;          m_materialEmi[1] = 0.0f;
m_materialEmi[2] = 0.0f;          m_materialEmi[3] = 1.0f;

m_matshininess = 0.0f;
```

(2) 编辑 Color/Material 对话框

打开 Color/Material 对话框,在原有的 Color 组下按表 7-8 给出的各控件信息增加诸控件,如图 7-6 所示。

表 7-8

控 件	ID	Caption
组	IDC_STATIC	Directional
静态文本框	IDC_STATIC	R:
静态文本框	IDC_STATIC	G:
静态文本框	IDC_STATIC	B:
静态文本框	IDC_STATIC	Ambient:
静态文本框	IDC_STATIC	Diffuse:
静态文本框	IDC_STATIC	Specular:
静态文本框	IDC_STATIC	Shininess:
复选按钮	IDC_MATERIAL_FLAG	On/Off Material
编辑框	IDC_M_AMBIENT_R	
编辑框	IDC_M_AMBIENT_G	
编辑框	IDC_M_AMBIENT_B	
编辑框	IDC_M_DIFFUSE_R	
编辑框	IDC_M_DIFFUSE_G	
编辑框	IDC_M_DIFFUSE_B	
编辑框	IDC_M_SPECULAR_R	
编辑框	IDC_M_SPECULAR_G	
编辑框	IDC_M_SPECULAR_B	
编辑框	IDC_M_EMISSION_R	
编辑框	IDC_M_EMISSION_G	
编辑框	IDC_M_EMISSION_B	
编辑框	IDC_M_SHININESS	

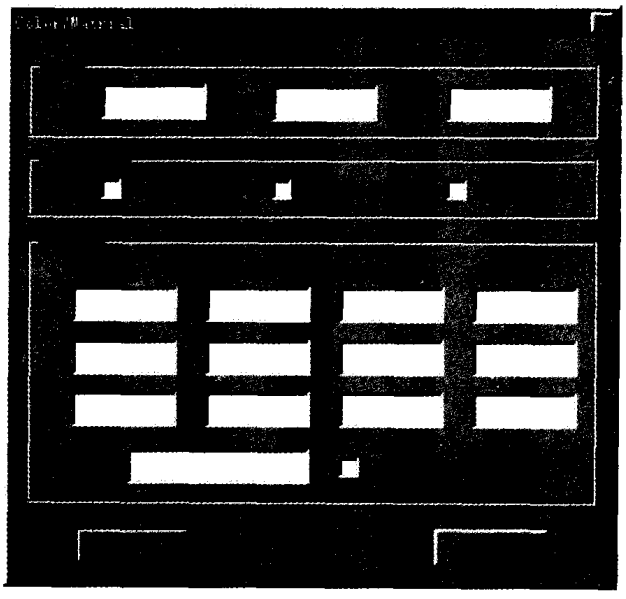

图 7-6　颜色材质对话框

按表 7-9 为上述新控件添加绑定成员变量。

表 7-9

控　件	变　量	类　型
IDC_M_AMBIENT_R	m_matAmbientR	float
IDC_M_AMBIENT_G	m_matAmbientG	float
IDC_M_AMBIENT_B	m_matAmbientB	float
IDC_M_DIFFUSE_R	m_matDiffuseR	float
IDC_M_DIFFUSE_G	m_matDiffuseG	float
IDC_M_DIFFUSE_B	m_matDiffuseB	float
IDC_M_SPECULAR_R	m_matSpecularR	float
IDC_M_SPECULAR_G	m_matSpecularG	float
IDC_M_SPECULAR_B	m_matSpecularB	float
IDC_M_EMISSION_R	m_matEmissionR	float
IDC_M_EMISSION_G	m_matEmissionG	float
IDC_M_EMISSION_B	m_matEmissionB	float
IDC_M_SHININESS	m_matShininess	float
IDC_MATERIAL_FLAG	m_bMaterialOnOff	BOOL

(3) 编辑 OnColorMaterial()函数

在语句 dlg.m_maskBlue = m_bMaskBlue;后添加以下代码：

```
// 材质
dlg.m_matAmbientR = m_materialAmb[0];
dlg.m_matAmbientG = m_materialAmb[1];
```

```
dlg.m_matAmbientB = m_materialAmb[2];

dlg.m_matDiffuseR = m_materialDif[0];
dlg.m_matDiffuseG = m_materialDif[1];
dlg.m_matDiffuseB = m_materialDif[2];

dlg.m_matSpecularR = m_materialSpe[0];
dlg.m_matSpecularG = m_materialSpe[1];
dlg.m_matSpecularB = m_materialSpe[2];

dlg.m_matEmissionR = m_materialEmi[0];
dlg.m_matEmissionG = m_materialEmi[1];
dlg.m_matEmissionB = m_materialEmi[2];

dlg.m_matShininess = m_matshininess;
dlg.m_bMaterialOnOff = m_bMaterial;
```

在语句 m_bMaskBlue = dlg.m_maskBlue;后添加以下代码：

```
m_materialAmb[0] = dlg.m_matAmbientR;
m_materialAmb[1] = dlg.m_matAmbientG;
m_materialAmb[2] = dlg.m_matAmbientB;

m_materialDif[0] = dlg.m_matDiffuseR;
m_materialDif[1] = dlg.m_matDiffuseG;
m_materialDif[2] = dlg.m_matDiffuseB;

m_materialSpe[0] = dlg.m_matSpecularR;
m_materialSpe[1] = dlg.m_matSpecularG;
m_materialSpe[2] = dlg.m_matSpecularB;

m_materialEmi[0] = dlg.m_matEmissionR;
m_materialEmi[1] = dlg.m_matEmissionG;
m_materialEmi[2] = dlg.m_matEmissionB;

m_matshininess = dlg.m_matShininess;
m_bMaterial = dlg.m_bMaterialOnOff;
```

该函数的最后结果如下：

```
void CLightView::OnColorMaterial()
{
    // TODO: Add your command handler code here
    CLightMaterial dlg;

    dlg.m_colorRed = m_colorRed;
```

```
dlg.m_colorGreen = m_colorGreen;
dlg.m_colorBlue = m_colorBlue;
dlg.m_maskRed = m_bMaskRed;
dlg.m_maskGreen = m_bMaskGreen;
dlg.m_maskBlue = m_bMaskBlue;

//材质
dlg.m_matAmbientR = m_materialAmb[0];
dlg.m_matAmbientG = m_materialAmb[1];
dlg.m_matAmbientB = m_materialAmb[2];

dlg.m_matDiffuseR = m_materialDif[0];
dlg.m_matDiffuseG = m_materialDif[1];
dlg.m_matDiffuseB = m_materialDif[2];

dlg.m_matSpecularR = m_materialSpe[0];
dlg.m_matSpecularG = m_materialSpe[1];
dlg.m_matSpecularB = m_materialSpe[2];

dlg.m_matEmissionR = m_materialEmi[0];
dlg.m_matEmissionG = m_materialEmi[1];
dlg.m_matEmissionB = m_materialEmi[2];

dlg.m_matShininess = m_matShininess;
dlg.m_bMaterialOnOff = m_bMaterial;

int responeDlg = dlg.DoModal();
if(responeDlg == IDOK)
{
    m_colorRed = dlg.m_colorRed;
    m_colorGreen = dlg.m_colorGreen;
    m_colorBlue = dlg.m_colorBlue;
    m_bMaskRed = dlg.m_maskRed;
    m_bMaskGreen = dlg.m_maskGreen;
    m_bMaskBlue = dlg.m_maskBlue;

    m_materialAmb[0] = dlg.m_matAmbientR;
    m_materialAmb[1] = dlg.m_matAmbientG;
    m_materialAmb[2] = dlg.m_matAmbientB;

    m_materialDif[0] = dlg.m_matDiffuseR;
    m_materialDif[1] = dlg.m_matDiffuseG;
    m_materialDif[2] = dlg.m_matDiffuseB;
```

```
            m_materialSpe[0] = dlg.m_matSpecularR;
            m_materialSpe[1] = dlg.m_matSpecularG;
            m_materialSpe[2] = dlg.m_matSpecularB;

            m_materialEmi[0] = dlg.m_matEmissionR;
            m_materialEmi[1] = dlg.m_matEmissionG;
            m_materialEmi[2] = dlg.m_matEmissionB;

            m_matShininess = dlg.m_matShininess;
            m_bMaterial = dlg.m_bMaterialOnOff;
            Invalidate();
        }
}
```

(4) 添加保护成员函数

添加一个保护成员函数 CreateMaterial(),该函数用来定义并启动材质。编辑如下:

```
void CLightView::CreateMaterial()
{
    glMaterialfv(GL_FRONT,GL_AMBIENT,m_materialAmb);
    glMaterialfv(GL_FRONT,GL_DIFFUSE,m_materialDif);
    glMaterialfv(GL_FRONT,GL_SPECULAR,m_materialSpe);
    glMaterialfv(GL_FRONT,GL_EMISSION,m_materialEmi);
    glMaterialf(GL_FRONT,GL_SHININESS,m_matShininess);
}
```

(5) 编辑 DrawScene()函数

在这里,我们用材质取代物体的单一颜色。也就是说,要么物体有材质,要么物体是单一颜色的。编辑并修改如下:

```
    if(m_bMaterial)
    {
        CreateMaterial();
        CreateLights();
    }
```

在 glFinish();语句前添加:

```
    glDisable(GL_LIGHTING);
```

以关闭光照。

最后,DrawScene()函数如下:

```
void CLightView::DrawScene()
{
    glClearColor(1.0f,1.0f,1.0f,1.0f);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

```
    glColor3f(0.2f, 0.5f, 0.8f);

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glTranslatef(0.0f,0.0f, -4.5f);

    glColorMask(! m_bMaskRed,! m_bMaskGreen,! m_bMaskBlue,GL_TRUE);
    glColor3f(m_colorRed,m_colorGreen,m_colorBlue);

    if(m_bMaterial)
    {
        CreateMaterial();
        CreateLights();
    }

    switch(m_selectObject)
    {
        case Cube:          DrawCube();     break;
        case Sphere:        DrawSphere();   break;
    }

    glDisable(GL_LIGHTING);

    glFinish();
}
```

当没有光照且材质关闭的情况下,物体的颜色就变为单一颜色了,而且也没有立体感,就像我们在第 2 节中所得到的结果一样。

(6) 将 Parameter 菜单的 Color 项的 Caption 更为 C&olor/Material...

(7) 运行程序

到这一步为止,我们已经通过编程实现了对光线和材质的实时模拟。这可是成百上千的物理学家、数学家、计算机科学家等多年辛勤研究的成果。此时,你可以通过菜单分别打开两个对话框,任意输入光和材质两个对话框中的各个参数,注意输入值和程序输出的结果是否一致。当启动材质时,OpenGL 的 glColor()函数将自动无效,但颜色屏蔽仍然有用。请注意,盒子仍是我之巍然不动! 这究竟是什么原因?

7.6 盒子就是盒子——法线的作用

对于颜色填充和材质的定义,区分多边形的前后面十分重要。在第 4 章,我们采用时针顺序来定义多边形的前后面,但是对于一个由多个顶点组成的面按这种方式来定义前面或后面显然是单调而又低效率的。OpenGL 可以通过法线来定义多边形的前后面。

有关法线的知识在高等数学或微分几何中都已经学过。在较薄的白纸上画一个带箭头的逆时针指向的圆圈。这时它的转向是逆时针的。把纸翻转过来,对光看一看,圆圈的转向变为

顺时针方向。假定绘有圆圈的面是纸的前面，这时，我们是从上往下看的。也就是说，纸平面的法线垂直纸面向上，其方向是指向眼睛的，我们就是从这个方向向下看它是逆时针的。当纸翻过来之后，我们是从前面的法线的负方向去看同一个圆圈的。要确定一个面的法线，只需知道这个面的3个顶点。当然这3个顶点也应该是按顺序排列的，这个小小的要求只是为了计算的方便。法线的方向则遵循右手规则，即4个手指的方向为顶点的顺序方向，大拇指所指的方向即为法线方向，如图7-7。位于法线一侧的面则是前面，背离法线的面则是后面。这种定义更加确切，因为无论这个面在空间的位置如何，其前后面均可采用这种方法来定义，绝不产生二义性。

一个面的法线对于OpenGL光照是十分必要的。通过法线矢量，OpenGL就知道光的入射角度，从而根据光照模型决定怎样对物体进行光照。法线对于曲面的生成、纹理映射等等都是必不可少的信息。

下面给出给定3个顺序顶点的法线矢量的计算程序。

为CLightView类添加一个保护成员函数：CalculateNormal()，编辑如下：

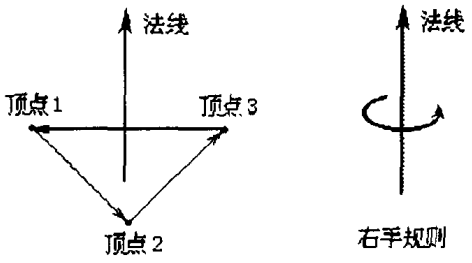

图7-7　定义法线

```
void CLightView::CalculateNormal(GLdouble * dVertex1, GLdouble * dVertex2,
                                 GLdouble * dVertex3, GLdouble * dNormal)
{
    GLdouble dVector1[3],dVector2[3];
    dVector1[0] = dVertex2[0] - dVertex1[0];
    dVector1[1] = dVertex2[1] - dVertex1[1];
    dVector1[2] = dVertex2[2] - dVertex1[2];
    dVector2[0] = dVertex3[0] - dVertex1[0];
    dVector2[1] = dVertex3[1] - dVertex1[1];
    dVector2[2] = dVertex3[2] - dVertex1[2];

    dNormal[0] = dVector1[1] * dVector2[2] - dVector1[2] * dVector2[1];
    dNormal[1] = dVector1[2] * dVector2[0] - dVector1[0] * dVector2[2];
    dNormal[2] = dVector1[0] * dVector2[1] - dVector1[1] * dVector2[0];

    double dNormalLength = sqrt(dNormal[0] * dNormal[0]
                                + dNormal[1] * dNormal[1] + dNormal[2] * dNormal[2]);
    if(dNormalLength! = 0.0)
```

```
    {
        dNormal[0] = dNormal[0]/dNormalLength;
        dNormal[1] = dNormal[1]/dNormalLength;
        dNormal[2] = dNormal[2]/dNormalLength;
    }
    else
    {
        dNormal[0] = 0.0;
        dNormal[1] = 0.0;
        dNormal[2] = 1.0;
    }
}
```

位于平面上的顶点其法线矢量都是相同的,但对于曲面的顶点,其法线矢量就各不相同。OpenGL 可以为每个顶点都定义法线:

```
glNormal3f(x,y,z);
```

该语句定义后继顶点的法线矢量为(x,y,z)。该函数共有 10 种不同的形式。

一般地,一个法线矢量并不是单位矢量。OpenGL 只处理归一化的法线矢量。

```
glEnable(GL_NORMALIZE);
```

使法线矢量归一化。法线矢量归一化的目的是使法线矢量位于 OpenGL 缺省的网格系统之内。单位矢量是归一化矢量。

下面,我们重新编辑 DrawCube()函数,给盒子的每个面增加法线矢量。打开 LightView.cpp 文件,重新编辑 DrawCube()函数,把黑体语句加入到该函数中:

```
void CLightView::DrawCube()
{
    double dPoint1[] = {-0.5,0.5,0.5};
    double dPoint2[] = {-0.5,-0.5,0.5};
    double dPoint3[] = {0.5,-0.5,0.5};
    double dNormal[3];

    glPushMatrix();
        glRotatef(25.0f,1.0f,0.0f,0.0f);
        glRotatef(20.0f,0.0f,1.0f,0.0f);
        glScalef(1.2f,1.2f,1.2f);

        ///////前面
        CalculateNormal(dPoint1,dPoint2,dPoint3,dNormal);
        glBegin(GL_POLYGON);
            glNormal3dv(dNormal);
            glVertex3f(-0.5f,0.5f,0.5f);
            glVertex3f(-0.5f,-0.5f,0.5f);
```

```
        glVertex3f(0.5f, -0.5f,0.5f);
        glVertex3f(0.5f,0.5f,0.5f);
glEnd();

///////右面
dPoint1[0] = 0.5;    dPoint1[1] = 0.5;     dPoint1[2] = 0.5;
dPoint2[0] = 0.5;    dPoint2[1] = -0.5;    dPoint2[2] = 0.5;
dPoint3[0] = 0.5;    dPoint3[1] = -0.5;    dPoint3[2] = -0.5;
CalculateNormal(dPoint1,dPoint2,dPoint3,dNormal);

glBegin(GL_POLYGON);
        glNormal3dv(dNormal);
        glVertex3f(0.5f,0.5f,0.5f);
        glVertex3f(0.5f, -0.5f,0.5f);
        glVertex3f(0.5f, -0.5f, -0.5f);
        glVertex3f(0.5f,0.5f, -0.5f);
glEnd();

////////后面
dPoint1[0] = 0.5;    dPoint1[1] = 0.5;     dPoint1[2] = -0.5;
dPoint2[0] = 0.5;    dPoint2[1] = -0.5;    dPoint2[2] = -0.5;
dPoint3[0] = -0.5;   dPoint3[1] = -0.5;    dPoint3[2] = -0.5;
CalculateNormal(dPoint1,dPoint2,dPoint3,dNormal);

glBegin(GL_POLYGON);
        glNormal3dv(dNormal);
        glVertex3f(0.5f,0.5f, -0.5f);
        glVertex3f(0.5f, -0.5f, -0.5f);
        glVertex3f( -0.5f, -0.5f, -0.5f);
        glVertex3f( -0.5f,0.5f, -0.5f);
glEnd();

//////左面
dPoint1[0] = -0.5;   dPoint1[1] = 0.5;     dPoint1[2] = -0.5;
dPoint2[0] = -0.5;   dPoint2[1] = -0.5;    dPoint2[2] = -0.5;
dPoint3[0] = -0.5;   dPoint3[1] = -0.5;    dPoint3[3] = 0.5;
CalculateNormal(dPoint1,dPoint2,dPoint3,dNormal);

glBegin(GL_POLYGON);
        glNormal3dv(dNormal);
        glVertex3f( -0.5f,0.5f, -0.5f);
        glVertex3f( -0.5f, -0.5f, -0.5f);
        glVertex3f( -0.5f, -0.5f,0.5f);
        glVertex3f( -0.5f,0.5f,0.5f);
```

```
    glEnd();

    ////////底面

    dPoint1[0] = -0.5;  dPoint1[1] = -0.5;    dPoint1[2] = 0.5;
    dPoint2[0] = -0.5;  dPoint2[1] = -0.5;    dPoint2[2] = -0.5;
    dPoint3[0] = 0.5;   dPoint3[1] = -0.5;    dPoint3[3] = -0.5;
    CalculateNormal(dPoint1,dPoint2,dPoint3,dNormal);

    glBegin(GL_POLYGON);
        glNormal3dv(dNormal);
        glVertex3f(0.5f, -0.5f,0.5f);
        glVertex3f(0.5f, -0.5f,0.5f);
        glVertex3f(0.5f, -0.5f, -0.5f);
        glVertex3f(0.5f, -0.5f, -0.5f);
    glEnd();

    ///////上面
    dPoint1[0] = -0.5;  dPoint1[1] = 0.5;     dPoint1[2] = -0.5;
    dPoint2[0] = -0.5;  dPoint2[1] = 0.5;     dPoint2[2] = 0.5;
    dPoint3[0] = 0.5;   dPoint3[1] = 0.5;     dPoint3[3] = 0.5;
    CalculateNormal(dPoint1,dPoint2,dPoint3,dNormal);

    glBegin(GL_POLYGON);
        glNormal3dv(dNormal);
        glVertex3f( -0.5f,0.5f, -0.5f);
        glVertex3f( -0.5f,0.5f,0.5f);
        glVertex3f(0.5f,0.5f,0.5f);
        glVertex3f(0.5f,0.5f, -0.5f);
    glEnd();

  glPopMatrix();
}
```

重新运行程序并绘制盒子。怎么样？盒子就是盒子！如图 7-8 所示。

7.7 明暗着色处理

7.7.1 明暗着色处理

计算机图形学中的明暗处理是对艺术家的绘制技艺的模拟。艺术家(特别是油画艺术家)往往不用一种颜色去绘制同一个区域，各个绘制区域间的颜色过渡、整体明暗效果十分自然。

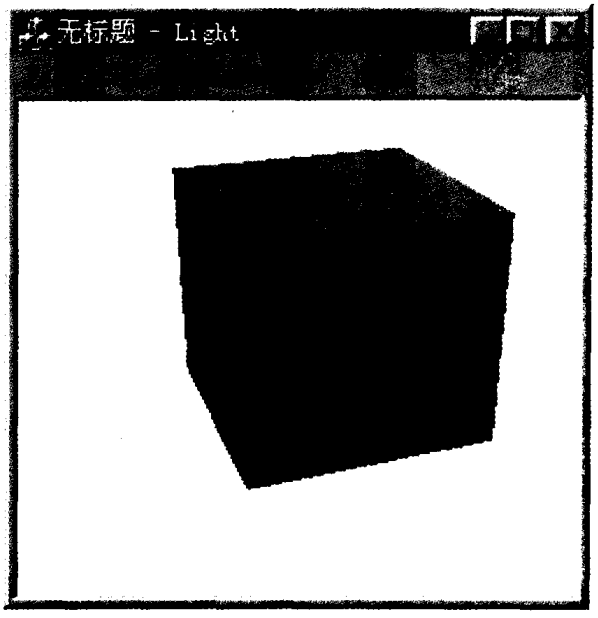

图 7-8　光照盒子

计算机图像处理不能超越艺术思想,但可以超越艺术手法。绝大多数的三维物体都是由多边形所构成的,必须进行适当的着色处理看起来才像一个三维物体。OpenGL 可以对场景中的物体进行平面明暗着色处理和光滑明暗着色处理。

glShadeModel(GL _ FLAT);

告诉 OpenGL 对场景中的物体进行平面明暗着色处理。平面明暗着色处理就是用一种单一颜色去填充一个多边形。多边形的颜色是由其顶点的颜色决定的。对于单个顶点,平面明暗着色处理和光滑明暗着色处理没有区别。对于直线段,其颜色由第一个顶点的颜色决定。对于折线,除第一条直线段外,新增顶点的颜色决定了新增线段的颜色。对于单个多边形,其填充色是由第一个顶点的颜色决定的。对于三角形片和三角形扇,新增三角形的填充色由新增顶点的颜色决定。平面明暗着色处理方法虽然会产生不真实的效果,不过它非常适合于快速成像及速度重于图片精度的场合。

glShadeModel(GL _ SMOOTH);

这是默认设置,表示对场景中的物体进行光滑明暗着色处理。光滑明暗着色处理不再以单一颜色去填充多边形,其颜色填充与每个顶点的颜色都有关。多边形内点的颜色是由顶点的颜色经过线性插值计算获得的,而每个顶点的颜色仍保持不变。这种着色效果比平面明暗着色好得多。因为它对三维模型上的每个顶点的颜色进行平滑、融合处理,将多边形内部填上顺滑的渐变色,其外观有更强烈的实时感和立体感。这就是著名的 Gouraud 明暗着色处理方法。经过光滑明暗着色处理,一个多边形的颜色就会平滑过渡,靠近顶点的区域,顶点色就越突出。不过这种着色处理方法较慢。

再次强调,OpenGL 的几何图元是顶点。与顶点相关的信息有顶点的颜色、法线矢量、纹理坐标等。这些信息是在 glBegin()/glEnd()函数对中被送入 OpenGL 的。在该对中可以应用以下函数:

glCallList(),　　glCallLists(),

glColor(),　　glEdgeFlag(),

glEvalCoord(), glIndex(),
glMaterial(), glNormal(),
glTexCoord(), glVertex()。

7.7.2 平滑盒子

为了不增加编程的复杂性,我们把这一部分单独处理,即,不与材质和光照等已有程序关联起来。

(1) 编辑 LightView.h 文件

在 LightView.h 文件的 enum{Cube,Sphere}中增加一项 SmoothCube:

```
enum{Cube,Sphere, SmoothCube };
```

(2) 编辑 Select 菜单

在 Select 菜单中新增一个菜单项,其 ID 号为 IDM_SMOOTH_CUBE,并在其 Caption 框内键入 S&mooth Cube。

(3) 响应菜单

为 IDM_SMOOTH_CUBE 增加 Windows 消息,并响应如下:

```
void CLightView::OnSmoothCube()
{
    // TODO: Add your command handler code here
    m_selectObject = SmoothCube;
    Invalidate();
}

void CLightView::OnUpdateSmoothCube(CCmdUI * pCmdUI)
{
    // TODO: Add your command update UI handler code here
    if( m_selectObject == SmoothCube)
        pCmdUI->SetCheck(TRUE);
    else
        pCmdUI->SetCheck(FALSE);
}
```

(4) 添加保护成员函数

添加保护成员函数 DrawSmoothCube(),编辑如下:

```
void CLightView::DrawSmoothCube()
{
    glDisable(GL_LIGHTING);

    glPushMatrix();
        glRotatef(25.0f,1.0f,0.0f,0.0f);
        glRotatef(45.0f,0.0f,1.0f,0.0f);
```

```
glScalef(1.2f,1.2f,1.2f);

glBegin(GL_QUAD_STRIP);
        glColor3f(1.0f, 0.0f, 0.0f);
        glVertex3f(-0.5f, 0.5f, 0.5f);
        glColor3f(1.0f, 0.0f, 1.0f);
        glVertex3f(-0.5f, -0.5f, 0.5f);
        glColor3f(1.0f, 1.0f, 1.0f);
        glVertex3f(0.5f, 0.5f, 0.5f);
        glColor3f(1.0f, 1.0f, 0.0f);
        glVertex3f(0.5f, -0.5f, 0.5f);
        glColor3f(0.0f, 1.0f, 1.0f);
        glVertex3f(0.5f, 0.5f, -0.5f);
        glColor3f(0.0f, 1.0f, 0.0f);
        glVertex3f(0.5f, -0.5f, -0.5f);
        glColor3f(0.0f, 0.0f, 1.0f);
        glVertex3f(-0.5f, 0.5f, -0.5f);
        glColor3f(0.0f, 0.0f, 0.0f);
        glVertex3f(-0.5f, -0.5f, -0.5f);
        glColor3f(1.0f, 0.0f, 1.0f);
        glVertex3f(-0.5f, 0.5f, 0.5f);
        glColor3f(1.0f, 0.0f, 0.0f);
        glVertex3f(-0.5f, -0.5f, 0.5f);
glEnd();

glBegin(GL_QUADS);
        glColor3f(1.0f, 0.0f, 1.0f);
        glVertex3f(-0.5f, 0.5f, 0.5f);

        glColor3f(1.0f, 1.0f, 1.0f);
        glVertex3f(0.5f, 0.5f, 0.5f);

        glColor3f(0.0f, 1.0f, 1.0f);
        glVertex3f(0.5f, 0.5f, -0.5f);

        glColor3f(0.0f, 0.0f, 1.0f);
        glVertex3f(-0.5f, 0.5f, -0.5f);
glEnd();

glBegin(GL_QUADS);
        glColor3f(1.0f, 0.0f, 0.0f);
        glVertex3f(-0.5f, -0.5f, 0.5f);
        glColor3f(1.0f, 1.0f, 0.0f);
        glVertex3f(0.5f, -0.5f, 0.5f);
```

```
            glColor3f(0.0f, 1.0f, 0.0f);
            glVertex3f(0.5f, -0.5f, -0.5f);
            glColor3f(0.0f, 0.0f, 0.0f);
            glVertex3f(-0.5f, -0.5f, -0.5f);
        glEnd();
    glPopMatrix();

    glEnable(GL_LIGHTING);
}
```

(5) 编辑 DrawScene()函数

在语句 glColor3f(m_colorRed,m_colorGreen,m_colorBlue); 之后添加:

```
    if(m_selectObject==SmoothCube)
    {
        DrawSmoothCube(); glFinish(); return;
    }
```

最后,DrawScene()函数如下:

```
void CLightView::DrawScene()
{
    glClearColor(1.0f,1.0f,1.0f,1.0f);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glColor3f(0.2f, 0.5f, 0.8f);

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glTranslatef(0.0f,0.0f,-4.5f);

    glColorMask(!m_bMaskRed,!m_bMaskGreen,!m_bMaskBlue,GL_TRUE);
    glColor3f(m_colorRed,m_colorGreen,m_colorBlue);

    if(m_selectObject==SmoothCube)
    {
        DrawSmoothCube(); glFinish(); return;
    }

    if(m_bMaterial)
    {
        CreateMaterial();
        CreateLights();
    }
    switch(m_selectObject)
    {
        case Cube:          DrawCube();     break;
```

```
        case Sphere:        DrawSphere();    break;
    }

    glDisable(GL_LIGHTING);
    glFinish();
}
```

(6) 运行程序

编译连接并运行程序,执行菜单 Select \ Smooth Cube,绘制平滑盒子,如图 7-9 所示。可见,在没有进行光照的情况下,平滑处理的盒子却有了立体感!

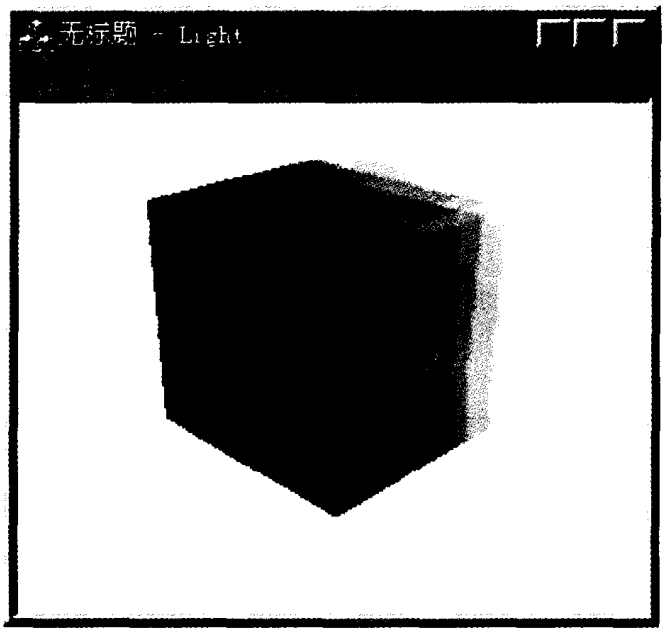

图 7-9 光顺盒子

7.8 深度测试

为了正确地进行消隐,OpenGL 必须知道各物体间的相对位置关系。例如,位于视点前面的物体会遮住后面的物体,因此,必须进行深度测试。

深度测试的通用方法是 Z 缓存(Z-buffer)技术。Z-buffer 记录在每个像素点上距离观察点最近的物体,由此决定表面的可见性。Z-buffer 是一个位平面,它与帧存有关,用于存放窗口中从近剪切面至每个像素的距离。如果深度测试条件是小于条件,在 Z-buffer 模式中,则即将画的像素 z 坐标同那个像素处已经绘制的几何体的 z 坐标进行比较,如果即将画的值表明新几何体比已经存在的几何体距离观察点近,那么分别存储于颜色缓存和 Z-buffer 中原来的颜色值和 Z 值就会被新值所代替。Z-buffer 也叫深度缓存。这种过程称为深度测试。与颜色缓存一样,OpenGL 可以对深度缓存进行操作:

(1) 清除深度缓存:

```
glClear(GL_DEPTH_BUFFER_BIT);
```

(2) 设置清除值:

```
glClearDepth(1.0);
```

该函数采用 GLclampd 类型为参数。参数值的范围为[0,1]。也可以利用 glGet()函数来获得这个范围。glClear()函数和 glClearDepth()函数我们已经用过(见 Init()函数和 DrawScene()函数)。

(3) 屏蔽深度缓存:

```
glDepthMask(GL_TRUE); 可以写深度缓存;
glDepthMask(GL_FALSE); 则禁止写深度缓存。
```

(4) 启动和关闭深度测试:

```
glEnable(GL_DEPTH_TEST);
glDisable(GL_DEPTH_TEST);
```

(5) 确定测试条件:

```
glDepthFunc(GL_LESS);
```

该函数有一个参数,它只能采用表 7-10 中的一个常量:

表 7-10

常 量	含 义
GL_LESS	小于则通过
GL_LEQUAL	小于或等于则通过
GL_EQUAL	相等则通过
GL_GEQUAL	大于或等于则通过
GL_GREATER	大于则通过
GL_NOTEQUAL	不相等则通过
GL_NEVER	不通过
GL_ALWAYS	总是通过

上表的含义是:例如,GL_LESS,则表示在同一像素处,即将画的物体 z 坐标如果比已经存储的 z 坐标小就存储后者的 z 坐标,同时称后者通过测试。其他含义以此类推。这里的 z 坐标,是指物体到观察点的距离,因而是观察坐标。

(6) 确定深度范围

确定深度范围的函数是 glDepthRange(),其原型为:

```
void glDepthRange(Glclampd zNear,GLclampd zFar);
```

zNear 参数说明近剪切面怎样向窗口坐标映射,缺省值为 0.0;zFar 参数说明远剪切面怎样向窗口坐标映射,缺省值为 1.0。通过剪切和透视分割之后,坐标的范围就从 -1.0 到 1.0,即得到了深度的规格化设备坐标。glDepthRange()函数则建立了一个[-1.0,1.0]至[0.0,1.0]的线性映射,这种映射带来计算的方便,因为此时已经不需考虑实际深度。缺省情况下,

由于0.0代表近剪切面被映射到0.0,1.0代表近剪切面被映射到1.0,因此深度缓存得到最充分的利用。

7.9 光照模型

前面我们只讨论了如何定义光的基本属性。事实上,光对物体的照射是十分复杂的,感知光照效果与眼睛的位置也有很大的关系。在现实生活中,光作为一种物质,是要衰减的。因此,我们有必要对光的这些物理性质进行模拟。为此,需要建立光照模型。光照模型分为局部光照模型和整体光照模型。局部光照模型重点模拟光源引起的漫反射和镜面反射。著名的模型有 Phong 光照模型、Torrance-Sparrow 光照模型、Cook-Torrance 光照模型。整体光照模型一般采用光线跟踪技术,如 Whitted 光照模型。

OpenGL 通过 glLightModel()函数近似模拟光照。该函数的第一个参数为一个 OpenGL 常量,只能取如下三者之一:

GL_LIGHT_MODEL_AMBIENT,

GL_LIGHT_MODEL_LOCAL_VIEWER,

GL_LIGHT_MODEL_TWO_SIDE。

7.9.1 全局环境光

全局环境光不来自于任何指定的光源,就好像环境本身就是这种光。下面的语句可以为整个场景提供一盏环境光。

```
GLfloat fGlobalAmbient[] = {0.4f,0.4f,0.4f,1.0f};
glLightModelfv(GL_LIGHT_MODEL_AMBIENT, fGlobalAmbient);
```

环境光既不是点光源也不是无限光源,它总是均匀散射在空间的各个部分。

7.9.2 镜面高光

镜面高光与光源的镜面反射光、材质的镜面反射率、物体表面的位置以及视点的位置有关。从计算上讲,镜面高光的计算除了与光源属性、材质属性有关外,还与表面顶点的法线矢量、视点至光源方向、顶点至视点方向有关。

应该注意,OpenGL 在进行光照计算时,另外还采用了一个视点。这个视点并不是观察点,而只是用于简化光照计算。下面的语句可以对视点进行调整:

```
glLightModeli(GL_LIGHT_MODEL_LOCAL_VIEWER,GL_TRUE);
```

当第二个参数为0时,视点到任何顶点的方向都是相同的,即把视点当成一个无穷远点来近似模拟观察点对物体的观察,这也是默认情形。当第二个参数不为0时,视点与观察点是同一个点,则视点到各个顶点的方向是不相同的。这符合观察实际,因而模拟的高光更加逼真。

7.9.3 双面光照

OpenGL 既支持单面光照也支持双面光照。单面光照时，只取表面的指定面的材质属性进行光照计算。对于双面光照，OpenGL 取前后面的材质属性进行前后面光照计算。

下面的语句表示进行双面光照：

```
glLightModeli(GL_LIGHT_MODEL_TWO_SIDE,GL_TRUE);
```

取消双面光照的语句是：

```
glLightModeli(GL_LIGHT_MODEL_TWO_SIDE,GL_FALSE);
```

OpenGL 也能进行前面光照计算。

7.10 聚　　光

7.10.1 定义聚光

剧场多采用聚光，手电筒发出的光是聚光。聚光较之点光源和无限光源更具有艺术表现力。下面我们说明怎样定义聚光。

(1) 定义聚光属性

聚光也是一种光，因此需要定义其环境反射光、漫反射光和镜面反射光。例如：

```
GLfloat        m_spotAmb[4] = {0.0f,0.0f,0.0f,1.0f};
GLfloat        m_spotDif[4] = {0.0f,1.0f,1.0f,1.0f};
GLfloat        m_spotSpe[4] = {0.6f,0.6f,0.6f,1.0f};
GLfloat        m_spotPos[4] = {-1.0f,0.0f,1.0f,1.0f};
GLfloat        m_spotDir[4] = {1.0f,0.0f,1.0f};
GLfloat        m_spotCutoff = 40.0f;
GLfloat        m_spotExponent = 2.0f;

glLightfv(GL_LIGHT1,GL_AMBIENT,m_spotAmb);
glLightfv(GL_LIGHT1,GL_DIFFUSE,m_spotDif);
glLightfv(GL_LIGHT1,GL_SPECULAR,m_spotSpe);
glLightfv(GL_LIGHT1,GL_POSITION,m_spotPos);
glLightfv(GL_LIGHT1,GL_SPOT_DIRECTION,m_spotDir);
glLightf(GL_LIGHT1,GL_SPOT_CUTOFF,m_spotCutoff);
glLightf(GL_LIGHT1,GL_SPOT_EXPONENT,m_spotExponent);
```

m_spotAmb 定义聚光的环境反射光，m_spotDif 定义漫反射光，m_spotSpe 定义镜面反射光，m_spotPos 定义聚光光源位置，聚光光源必须是定向光源，m_spotPos 说明聚光方向。一个聚光必须指明其光照方向，这个方向即是光锥的中心轴线所在的方向。其缺省值为(0.0, 0.0, -1.0)，指向 z 轴负方向。m_spotCutoff 说明发散角的大小，是光锥母线与中心轴线的夹角，也即为发散角的一半。其取值范围为[0.0,90.0]或 180.0。当取 180.0 时，发散角就成

为360度,因而不再是一个聚光,而是一个点光源了,这是缺省设置。m_spotExponent说明聚光指数。我们都见过手电筒或是剧场用的聚光灯所照射出来的圆形灯光。光束边缘的模糊与清楚决定于手电筒的质量和对聚光灯的调整。对模糊的光束而言,中间的明亮圆圈叫聚光区(Hotspot)。在光束末端外围边缘和黑暗相接处为衰减区(Falloff)。聚光区和衰减区周围所造成的差异决定光束的强弱。比如,聚光区和衰减区大小相同,则会产生一个非常清楚的光束照射边缘。OpenGL采用聚光指数来控制聚光区和衰减区的大小。指数越大,光强越高。

(2) 说明光的衰减

随着光源与表面距离的增加,光源对表面产生的影响就会越小。OpenGL可以模拟这种衰减过程:

```
glLightf(GL_LIGHT1,GL_CONSTANT_ATTENUATION,2.0);
glLightf(GL_LIGHT1,GL_LINEAR_ATTENUATION,1.0);
glLightf(GL_LIGHT1,GL_QUADRATIC_ATTENUATION,0.8);
```

第一条语句说明光按常数衰减,第二条语句说明光线性衰减,第三条语句说明光二次衰减。衰减系数为:

$$F_a = \frac{1}{C_0 + C_1 \times distance + C_2 \times distance \times distance}$$

其中,distance表示顶点与光源的距离,C_0为常数衰减因子,C_1为线性衰减因子,C_2为二次衰减因子,它们分别由glLightf()函数的第三个参数提供。

对每个光照顶点,OpenGL都要计算衰减系数,而且点光源的环境光、漫反射光、镜面反射光都要衰减。

7.10.2 用聚光照射物体

下面我们为场景添加一盏聚光。

(1) 编辑对话框Light

加大对话框Light。拷贝对话框Light的所有控件,但除Ok和Cancel外。把组名更为Light2/Spot。对编辑框的ID号进行修改。例如:IDC_L_AMBIENT_R2更为IDC_S_AMBIENT_R。依次类推。按表7-11增加控件,如图7-10所示。

表7-11

控 件	变 量	类 型
静态文本框	IDC_STATIC	Direction/X:
静态文本框	IDC_STATIC	Direction/Y:
静态文本框	IDC_STATIC	Direction/Z:
静态文本框	IDC_STATIC	Cutoff:
编辑框	IDC_S_DIRECTION_X	
编辑框	IDC_S_DIRECTION_Y	
编辑框	IDC_S_DIRECTION_Z	
编辑框	IDC_SPOT_CUTOFF	

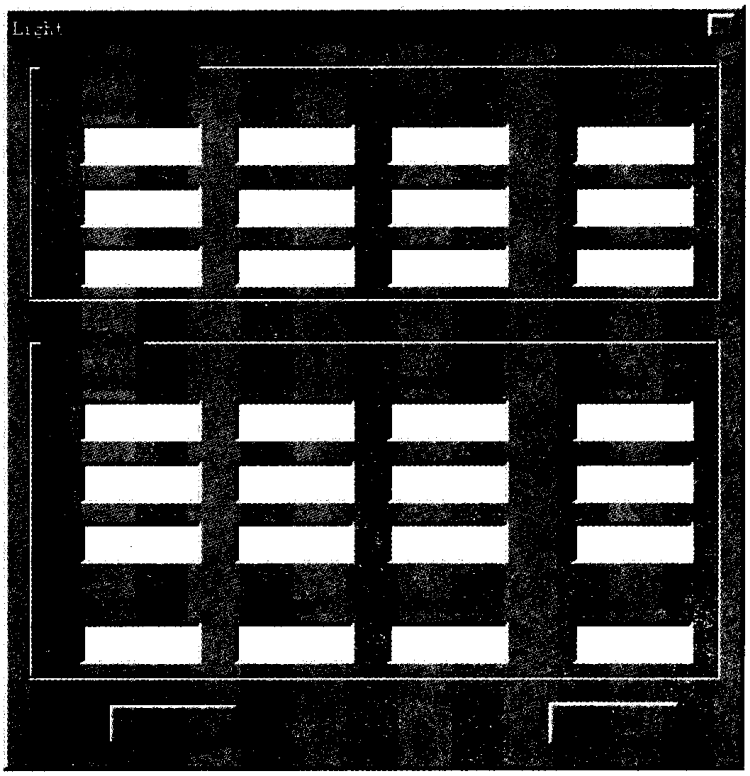

图 7-10　光照对话框

按表 7-12 为各控件增加变量。

表 7-12

控　件	变　量	类　型
IDC_S_AMBIENT_R	m_spotAmbientR	float
IDC_S_AMBIENT_G	m_spotAmbientG	float
IDC_S_AMBIENT_B	m_spotAmbientB	float
IDC_S_DIFFUSE_R	m_spotDiffuseR	float
IDC_S_DIFFUSE_G	m_spotDiffuseG	float
IDC_S_DIFFUSE_B	m_spotDiffuseB	float
IDC_S_SPECULAR_R	m_spotSpecularR	float
IDC_S_SPECULAR_G	m_spotSpecularG	float
IDC_S_SPECULAR_B	m_spotSpecularB	float
IDC_S_POSITION_X	m_spotPositonX	float
IDC_S_POSITION_Y	m_spotPositonY	float
IDC_S_POSITION_Z	m_spotPositonZ	float
IDC_S_DIRECTION_X	m_spotDirectionX	float
IDC_S_DIRECTION_Y	m_spotDirectionY	float
IDC_S_DIRECTION_Z	m_spotDirectionZ	float
IDC_SPOT_CUTOFF	m_spotCutoff	float

(2) 增加保护成员变量

为 CLightView 类增加保护成员变量：

```
GLfloat   m_spotAmb[4],          m_spotDif[4];
GLfloat   m_spotSpe[4],          m_spotPos[4];
GLfloat   m_spotDir[4],          m_spotCutoff;
```

并初始化：

```
m_spotAmb[0] = 0.2f;          m_spotAmb[1] = 0.2f;
m_spotAmb[2] = 0.2f;          m_spotAmb[3] = 1.0f;
m_spotDif[0] = 0.0f;          m_spotDif[1] = 1.0f;
m_spotDif[2] = 1.0f;          m_spotDif[3] = 1.0f;

m_spotSpe[0] = 0.6f;          m_spotSpe[1] = 0.6f;
m_spotSpe[2] = 0.6f;          m_spotSpe[3] = 1.0f;
m_spotPos[0] = -1.0f;         m_spotPos[1] = 0.0f;
m_spotPos[2] = 1.0f;          m_spotPos[3] = 1.0f;
m_spotDir[0] = 1.0f;          m_spotDir[1] = 0.0f;
m_spotDir[2] = -1.0f;         m_spotDir[3] = 1.0f;

m_spotCutoff = 40.0f;
```

(3) 编辑 OnLight()函数

在 lightDlg.m_lightPositionZ = m_lightPos[2]; 之后添加下述代码：

```
////////Spot
lightDlg.m_spotAmbientR = m_spotAmb[0];
lightDlg.m_spotAmbientG = m_spotAmb[1];
lightDlg.m_spotAmbientB = m_spotAmb[2];

lightDlg.m_spotDiffuseR = m_spotDif[0];
lightDlg.m_spotDiffuseG = m_spotDif[1];
lightDlg.m_spotDiffuseB = m_spotDif[2];

lightDlg.m_spotSpecularR = m_spotSpe[0];
lightDlg.m_spotSpecularG = m_spotSpe[1];
lightDlg.m_spotSpecularB = m_spotSpe[2];

lightDlg.m_spotPositionX = m_spotPos[0];
lightDlg.m_spotPositionY = m_spotPos[1];
lightDlg.m_spotPositionZ = m_spotPos[2];

lightDlg.m_spotDirectionX = m_spotDir[0];
lightDlg.m_spotDirectionY = m_spotDir[1];
lightDlg.m_spotDirectionZ = m_spotDir[2];
```

```
lightDlg.m_spotCutoff = m_spotCutoff;
```

在 m_lightPos[2] = lightDlg.m_lightPositionZ;之后添加下述代码：

```
//////////Spot
m_spotAmb[0] = lightDlg.m_spotAmbientR;
m_spotAmb[1] = lightDlg.m_spotAmbientG;
m_spotAmb[2] = lightDlg.m_spotAmbientB;

m_spotDif[0] = lightDlg.m_spotDiffuseR;
m_spotDif[1] = lightDlg.m_spotDiffuseG;
m_spotDif[2] = lightDlg.m_spotDiffuseB;

m_spotSpe[0] = lightDlg.m_spotSpecularR;
m_spotSpe[1] = lightDlg.m_spotSpecularG;
m_spotSpe[2] = lightDlg.m_spotSpecularB;

m_spotPos[0] = lightDlg.m_spotPositionX;
m_spotPos[1] = lightDlg.m_spotPositionY;
m_spotPos[2] = lightDlg.m_spotPositionZ;

m_spotDir[0] = lightDlg.m_spotDirectionX;
m_spotDir[1] = lightDlg.m_spotDirectionY;
m_spotDir[2] = lightDlg.m_spotDirectionZ;

m_spotCutoff = lightDlg.m_spotCutoff;
```

(4) 编辑 CreateLights()函数

在语句 glLightfv(GL_LIGHT0,GL_POSITION,m_lightPos);之后添加：

```
glLightfv(GL_LIGHT1,GL_AMBIENT,m_spotAmb);
glLightfv(GL_LIGHT1,GL_DIFFUSE,m_spotDif);
glLightfv(GL_LIGHT1,GL_SPECULAR,m_spotSpe);
glLightfv(GL_LIGHT1,GL_POSITION,m_spotPos);
glLightfv(GL_LIGHT1,GL_SPOT_DIRECTION,m_spotDir);
glLightf(GL_LIGHT1,GL_SPOT_CUTOFF,m_spotCutoff);
```

启动光照，在语句 glEnable(GL_LIGHT0);之后添加：

```
glEnable(GL_LIGHT1);
```

(5) 运行程序

编译连接并运行程序，在黄色球中出现了一个青色区域，如图 7-11 所示。立体感更强了，效果真不错！

(6) 编辑 Light 对话框，控制聚光的衰减

下面通过编程，实现对聚光的衰减控制。首先打开 Light 对话框，在 Light2/Spot 组内按表 7-13 添加 4 个复选按钮：

表 7-13

控　件	ID	Caption
复选框	IDC_S_CONSTANT	Constant
复选框	IDC_S_LINEAR	Linear
复选框	IDC_S_QUADRATIC	Quadratic
复选框	I DC_S__ONOFF	On/Off

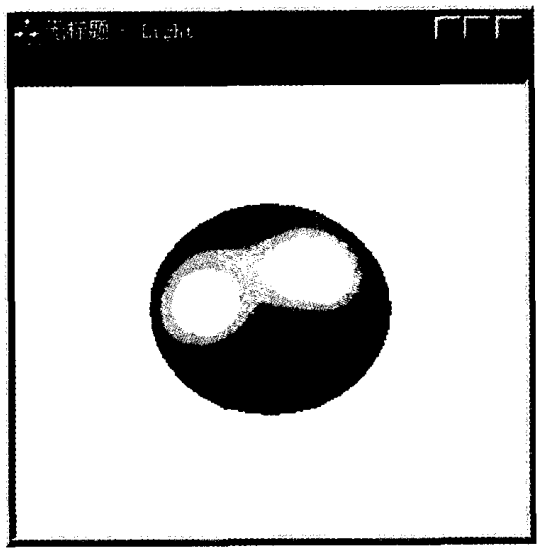

图 7-11　聚光的照射效果

按表 7-14 添加变量：

表 7-14

控　件	变　量	类　型
IDC_S_CONSTANT	m_bConstant	BOOL
IDC_S_LINEAR	m_bLinear	BOOL
IDC_S_QUADRATIC	m_bQuadratic	BOOL
I DC_S__ONOFF	m_bSpotOnOff	BOOL

(7) 添加保护成员变量

为 CLightView 类添加保护成员变量：

```
BOOL  m_bConstant;
BOOL  m_bLinear;
BOOL  m_bQuadratic;
BOOL  m_bSpotOnOff;
```

初始化如下：

```
m_bConstant = FALSE;
m_bLinear = FALSE;
m_bQuadratic = FALSE;
m_bSpotOnOff = TRUE;
```

(8) 编辑 OnLight()函数:

在语句 lightDlg.m_spotCutoff = m_spotCutoff;之后添加:

```
lightDlg.m_bConstant = m_bConstant;
lightDlg.m_bLinear = m_bLinear;
lightDlg.m_bQuadratic = m_bQuadratic;
lightDlg.m_bSpotOnOff = m_bSpotOnOff;
```

在语句 m_spotCutoff = lightDlg.m_spotCutoff;之后添加:

```
m_bConstant = lightDlg.m_bConstant;
m_bLinear = lightDlg.m_bLinear;
m_bQuadratic = lightDlg.m_bQuadratic;
m_bSpotOnOff = lightDlg.m_bSpotOnOff;
```

该函数的最终形式如下:

```
void CLightView::OnLight()
{
    // TODO: Add your command handler code here
    CLighting lightDlg;

    lightDlg.m_lightAmbientR = m_lightAmb[0];
    lightDlg.m_lightAmbientG = m_lightAmb[1];
    lightDlg.m_lightAmbientB = m_lightAmb[2];

    lightDlg.m_lightDiffuseR = m_lightDif[0];
    lightDlg.m_lightDiffuseG = m_lightDif[1];
    lightDlg.m_lightDiffuseB = m_lightDif[2];

    lightDlg.m_lightSpecularR = m_lightSpe[0];
    lightDlg.m_lightSpecularG = m_lightSpe[1];
    lightDlg.m_lightSpecularB = m_lightSpe[2];

    lightDlg.m_lightPositionX = m_lightPos[0];
    lightDlg.m_lightPositionY = m_lightPos[1];
    lightDlg.m_lightPositionZ = m_lightPos[2];

    ////////Spot
    lightDlg.m_spotAmbientR = m_spotAmb[0];
    lightDlg.m_spotAmbientG = m_spotAmb[1];
    lightDlg.m_spotAmbientB = m_spotAmb[2];
```

```
lightDlg.m_spotDiffuseR = m_spotDif[0];
lightDlg.m_spotDiffuseG = m_spotDif[1];
lightDlg.m_spotDiffuseB = m_spotDif[2];

lightDlg.m_spotSpecularR = m_spotSpe[0];
lightDlg.m_spotSpecularG = m_spotSpe[1];
lightDlg.m_spotSpecularB = m_spotSpe[2];

lightDlg.m_spotPositionX = m_spotPos[0];
lightDlg.m_spotPositionY = m_spotPos[1];
lightDlg.m_spotPositionZ = m_spotPos[2];

lightDlg.m_spotDirectionX = m_spotDir[0];
lightDlg.m_spotDirectionY = m_spotDir[1];
lightDlg.m_spotDirectionZ = m_spotDir[2];

lightDlg.m_spotCutoff = m_spotCutoff;

lightDlg.m_bConstant = m_bConstant;
lightDlg.m_bLinear = m_bLinear;
lightDlg.m_bQuadratic = m_bQuadratic;
lightDlg.m_bSpotOnOff = m_bSpotOnOff;

int responeDlg = lightDlg.DoModal();

if(responeDlg = = IDOK)
{
    m_lightAmb[0] = lightDlg.m_lightAmbientR;
    m_lightAmb[1] = lightDlg.m_lightAmbientG;
    m_lightAmb[2] = lightDlg.m_lightAmbientB;

    m_lightDif[0] = lightDlg.m_lightDiffuseR;
    m_lightDif[1] = lightDlg.m_lightDiffuseG;
    m_lightDif[2] = lightDlg.m_lightDiffuseB;

    m_lightSpe[0] = lightDlg.m_lightSpecularR;
    m_lightSpe[1] = lightDlg.m_lightSpecularG;
    m_lightSpe[2] = lightDlg.m_lightSpecularB;

    m_lightPos[0] = lightDlg.m_lightPositionX;
    m_lightPos[1] = lightDlg.m_lightPositionY;
    m_lightPos[2] = lightDlg.m_lightPositionZ;
```

```
        //////////Spot
        m_spotAmb[0] = lightDlg.m_spotAmbientR;
        m_spotAmb[1] = lightDlg.m_spotAmbientG;
        m_spotAmb[2] = lightDlg.m_spotAmbientB;

        m_spotDif[0] = lightDlg.m_spotDiffuseR;
        m_spotDif[1] = lightDlg.m_spotDiffuseG;
        m_spotDif[2] = lightDlg.m_spotDiffuseB;

        m_spotSpe[0] = lightDlg.m_spotSpecularR;
        m_spotSpe[1] = lightDlg.m_spotSpecularG;
        m_spotSpe[2] = lightDlg.m_spotSpecularB;

        m_spotPos[0] = lightDlg.m_spotPositionX;
        m_spotPos[1] = lightDlg.m_spotPositionY;
        m_spotPos[2] = lightDlg.m_spotPositionZ;

        m_spotDir[0] = lightDlg.m_spotDirectionX;
        m_spotDir[1] = lightDlg.m_spotDirectionY;
        m_spotDir[2] = lightDlg.m_spotDirectionZ;

        m_spotCutoff = lightDlg.m_spotCutoff;

        m_bConstant = lightDlg.m_bConstant;
        m_bLinear = lightDlg.m_bLinear;
        m_bQuadratic = lightDlg.m_bQuadratic;
        m_bSpotOnOff = lightDlg.m_bSpotOnOff;
        Invalidate();
    }
}
```

(9) 编辑 CreateLights()函数:

在语句 glLightf(GL_LIGHT1,GL_SPOT_CUTOFF,m_spotCutoff);之后添加:

```
if(m_bConstant)
    glLightf(GL_LIGHT1,GL_CONSTANT_ATTENUATION,2.0);
if(m_bLinear)
    glLightf(GL_LIGHT1,GL_LINEAR_ATTENUATION,1.0);
if(m_bQuadratic)
    glLightf(GL_LIGHT1,GL_QUADRATIC_ATTENUATION,0.8);
```

将语句 glEnable(GL_LIGHT1); 用以下判断语句来代替:

```
if(m_bSpotOnOff)
    glEnable(GL_LIGHT1);
else
```

```
        glDisable(GL_LIGHT1);
```

该函数的最终形式如下：

```
void CLightView::CreateLights()
{
    glLightfv(GL_LIGHT0,GL_AMBIENT,m_lightAmb);
    glLightfv(GL_LIGHT0,GL_DIFFUSE,m_lightDif);
    glLightfv(GL_LIGHT0,GL_SPECULAR,m_lightSpe);
    glLightfv(GL_LIGHT0,GL_POSITION,m_lightPos);

    glLightfv(GL_LIGHT1,GL_AMBIENT,m_spotAmb);
    glLightfv(GL_LIGHT1,GL_DIFFUSE,m_spotDif);
    glLightfv(GL_LIGHT1,GL_SPECULAR,m_spotSpe);
    glLightfv(GL_LIGHT1,GL_POSITION,m_spotPos);
    glLightfv(GL_LIGHT1,GL_SPOT_DIRECTION,m_spotDir);
    glLightf(GL_LIGHT1,GL_SPOT_CUTOFF,m_spotCutoff);

    if(m_bConstant)
        glLightf(GL_LIGHT1,GL_CONSTANT_ATTENUATION,2.0);
    if(m_bLinear)
        glLightf(GL_LIGHT1,GL_LINEAR_ATTENUATION,1.0);
    if(m_bQuadratic)
        glLightf(GL_LIGHT1,GL_QUADRATIC_ATTENUATION,0.8);

    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    if(m_bSpotOnOff)
        glEnable(GL_LIGHT1);
    else
        glDisable(GL_LIGHT1);
}
```

(10) 编辑 DrawScene()函数

在语句 switch(m_selectObject) {...}之后添加：

```
    glLightf(GL_LIGHT1,GL_CONSTANT_ATTENUATION,1.0);
    glLightf(GL_LIGHT1,GL_LINEAR_ATTENUATION,0.0);
    glLightf(GL_LIGHT1,GL_QUADRATIC_ATTENUATION,0.0);
```

最后，DrawScene()函数如下：

```
void CLightView::DrawScene()
{
    glClearColor(1.0f,1.0f,1.0f,1.0f);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glColor3f(0.2f, 0.5f, 0.8f);
```

```
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glTranslatef(0.0f,0.0f,-4.5f);
    //从下行添加绘制代码,使其位于glFinish()函数前:
    glColorMask(!m_bMaskRed,!m_bMaskGreen,!m_bMaskBlue,GL_TRUE);
    glColor3f(m_colorRed,m_colorGreen,m_colorBlue);

    if(m_selectObject==SmoothCube)
    {
        DrawSmoothCube(); glFinish(); return;
    }

    if(m_bMaterial)
    {
        CreateMaterial();
        CreateLights();
    }

    switch(m_selectObject)
    {
        case Cube:          DrawCube();     break;
        case Sphere:        DrawSphere();   break;
    }

    glLightf(GL_LIGHT1,GL_CONSTANT_ATTENUATION,1.0);
    glLightf(GL_LIGHT1,GL_LINEAR_ATTENUATION,0.0);
    glLightf(GL_LIGHT1,GL_QUADRATIC_ATTENUATION,0.0);

    glDisable(GL_LIGHTING);
    glFinish();
}
```

(11) 运行程序

再次编译连接并运行程序。看看聚光的衰减对物体的影响。请注意,衰减指数已由程序给定,运行时尚不能进行更改。当然可以在对话框中再加三个编辑控件来输入衰减指数。此时,可以关闭聚光,但第一盏灯总是开着的。输入各种数据,观察运行结果并进行理解。

7.11 创建逻辑调色盘

7.11.1 逻辑调色盘

Windows是一个多任务和多窗口的操作环境,每个应用程序或窗口对颜色的要求是不一

样的。如果所请求的颜色不能被设备表示,要保证应用程序的窗口显示正确的颜色就必须改变物理调色盘。Windows 的逻辑调色盘在使用颜色的应用程序和系统之间起缓冲作用。Windows 通过管理设备的系统调色盘而提供了一种与设备无关的方法来访问显示设备的颜色显示能力。应用程序通过创建一个或多个逻辑盘的方法来使用系统调色盘。当一个窗口请求系统使用逻辑调色盘中的某种颜色时,Windows 将逻辑调色盘中的项与系统调色盘中当前的项作完全匹配,对逻辑调色盘中不可能匹配的项,Windows 将它们设置到系统调色盘的尚未使用的项中。如果系统调色盘中所有项都被使用,Windows 将逻辑调色盘中其余的项匹配到系统调色盘中尽可能与之相近的表项上。为了协助这种匹配,Windows 在系统调色盘中保留了20 种静态颜色,通常被称为缺省调色盘,用它来匹配背景调色盘中的表项。Windows 总是先满足前台窗口的颜色要求,从而保证了活动窗口具有最好的颜色显示。

OpenGL 是为至少显示 64000 种颜色的显示设备系统所设计的。显示设备所支持的颜色越多,OpenGL 的图形描述能力就越强。在一个只能支持 256 种颜色的显示设备上输出 OpenGL 的图形将是极难看的。因而必须建立逻辑调色盘。对于一个真彩设备或不基于调色盘的设备,逻辑调色盘的作用就要小一些。在真彩设备上执行 OpenGL 的 RGBA 值主要受到设备通道的限制。对于 8 位设备,逻辑调色盘的颜色可以有 256 种,对于 16、24 或 32 位真彩设备,逻辑调色盘的颜色可达 4096 种。

Windows NT 和 Windows 95 的调色板是针对 256 色 8 位显示适配器的,是 GDI 的一部分。在标准的 Super-VGA 上,采用的是 8 位调色盘,红色和绿色被分配 3 位,蓝色则是 2 位。

在这里,我们的目标只是创建一个 8 位逻辑调色盘。

7.11.2 创建逻辑调色盘

逻辑调色盘是一个 GDI 对象,当被选入设备对象中后,它就指定了应用程序用于绘图的颜色。调色盘中的每一项指定了一种颜色。Windows.h 中的 LOGPALETTE 结构可以定义一个调色盘的信息:

```
type struct tagLOGPALETTE{
    WORD    palVersion;
    WORD    palNumEntries;
    PALETTEENTRY palPalEntry[1];
}LOGPALETTE;
```

其中,PALETTEENTRY 结构的定义为:

```
type struct tagPALETTEENTRY{
    BYTE    peRed;
    BYTE    peGreen;
    BYTE    peBlue;
    BYTE    peFlags;
}PALETTEENTRY;
```

这里,peFlags 指定如何使用调色盘表项,可为 NULL 和下述值之一:

PC_EXPLICT:指定逻辑调色盘表项的低位字是系统调色盘的索引,即应用程序要使用

显示设备调色盘中的内容。

PC_NOCOLLAPS:指定该颜色应放在系统调色盘中未被使用的表项中,而不是使它与一个调色盘中已有的颜色匹配。如果系统调色盘中没有未被使用的表项,Windows 将跳过 PC_NOCOLLAPS 旗帜,且该表项指定的颜色被正常地匹配。

PC_RESERVED:应用程序将动态地改变该项。因而,当该项已映射到系统调色盘上时,Windows 就不再将其他调色盘的颜色与该项进行匹配。从而,当应用程序改变了该项的内容时,不会影响其他应用程序。

当填充结构 LOGPALETTE 之后,就可以通过 CreatePalete()函数创建逻辑调色盘对象。该函数以指向结构类型为 LOGPALETTE 的变量的指针为参数,返回一个标识逻辑调色盘对象的句柄。

和处理其他的 GDI 对象一样,逻辑调色盘必须选入到将要使用逻辑调色盘的设备对象中。使用 SelectPalette()函数来实现:

HPALETTE SelectPalette(HDC hdc,HPALETTE hPalette,BOOL bForceBackGround);

其中,hdc 是一个设备对象的句柄,hPalette 是标识一个逻辑调色盘对象的句柄。bForceBackGround 为非零时,被选择的逻辑调色盘总是作为背景调色盘。如果为零,则仅当窗口有输入焦点时,被选择的逻辑调色盘才作为背景调色盘。该函数返回设备对象中被替换的原逻辑调色盘的句柄。

尽管逻辑调色盘已被选入到设备对象中,但在使用之前必须先实现该逻辑调色盘,可通过 RealizePalette()函数来完成:

int RealizePalette(HDC hdc);

此时,Windows 将逻辑调色盘与系统调色盘进行比较,并匹配相同的颜色。如果系统调色盘上有空间,Windows 将逻辑调色盘中未被匹配的颜色映射到系统调色盘中。最后,若仍有未被匹配的颜色,而又不能被映射到系统调色盘中,则 Windows 将这些颜色与系统调色盘中相近的颜色匹配。该函数返回逻辑调色盘中有多少项已被映射到系统调色盘。

一个逻辑调色盘对象不再被使用时,应该删除它。下面我们完成添加调色盘的任务。

(1) 添加变量

打开 LightView.cpp,在其头部定义如下变量:

```
unsigned char threeto8[8] =
{
    0, 0111>>1, 0222>>1, 0333>>1, 0444>>1, 0555>>1, 0666>>1, 0377
};

unsigned char twoto8[4] =
{
    0, 0x55, 0xaa, 0xff
};

unsigned char oneto8[2] = { 0, 255};

static int defaultOverride[13] =
{
```

```
    0, 3, 24, 27, 64, 67, 88, 173, 181, 236, 247, 164, 91
};

static PALETTEENTRY defaultPalEntry[20] =
{
    { 0,   0,   0,    0 },          { 0x80,0,   0,    0 },
    { 0,   0x80,0,    0 },          { 0x80,0x80,0,    0 },
    { 0,   0,   0x80, 0 },          { 0x80,0,   0x80, 0 },
    { 0,   0x80,0x80, 0 },          { 0xC0,0xC0,0xC0, 0 },

    { 192, 220, 192,  0 },          { 166, 202, 240,  0 },
    { 255, 251, 240,  0 },          { 160, 160, 164,  0 },

    { 0x80,0x80,0x80, 0 },          { 0xFF,0,   0,    0 },
    { 0,   0xFF,0,    0 },          { 0xFF,0xFF,0,    0 },
    { 0,   0,   0xFF, 0 },          { 0xFF,0,   0xFF, 0 },
    { 0,   0xFF,0xFF, 0 },          { 0xFF,0xFF,0xFF, 0 }
};
```

(2) 添加公有成员变量

为 CLightView 类添加公有成员变量:

```
CPalette    m_cPalette;
CPalette    *m_pOldPalette;
```

初始化:

```
m_pOldPalette = NULL;
```

(3) 添加成员函数

添加公有成员函数:ComponentFromIndex()和 CreateRGBPalette(),并编辑如下:

```
unsigned char CLightView::ComponentFromIndex(int i, UINT nbits, UINT shift)
{
    unsigned char val;
    val = (unsigned char) (i >> shift);

    switch (nbits)
      {
    case 1:
        val &= 0x1;
        return oneto8[val];
    case 2:
        val &= 0x3;
        return twoto8[val];
    case 3:
```

```
        val &= 0x7;
        return threeto8[val];

    default:
        return 0;
    }
}

void CLightView::CreateRGBPalette()
{
    PIXELFORMATDESCRIPTOR pfd;
    LOGPALETTE *pPal;
    int n, i;

    n = ::GetPixelFormat(m_pDC->GetSafeHdc());
    ::DescribePixelFormat(m_pDC->GetSafeHdc(), n, sizeof(pfd), &pfd);

    if (pfd.dwFlags & PFD_NEED_PALETTE)
    {
        n = 1 << pfd.cColorBits;
        pPal = (PLOGPALETTE) new char[sizeof(LOGPALETTE) +
            n * sizeof(PALETTEENTRY)];

        ASSERT(pPal != NULL);

        pPal->palVersion = 0x300;
        pPal->palNumEntries = n;

        for (i=0; i<n; i++)
        {
          pPal->palPalEntry[i].peRed =
                ComponentFromIndex(i, pfd.cRedBits, pfd.cRedShift);
          pPal->palPalEntry[i].peGreen =
                ComponentFromIndex(i, pfd.cGreenBits, pfd.cGreenShift);
          pPal->palPalEntry[i].peBlue =
                ComponentFromIndex(i, pfd.cBlueBits, pfd.cBlueShift);
          pPal->palPalEntry[i].peFlags = 0;
        }

        /* fix up the palette to include the default GDI palette */
        if ((pfd.cColorBits == 8)                           &&
          (pfd.cRedBits   == 3) && (pfd.cRedShift   == 0) &&
          (pfd.cGreenBits == 3) && (pfd.cGreenShift == 3) &&
          (pfd.cBlueBits  == 2) && (pfd.cBlueShift  == 6)
```

```
        )
        {
            for (i = 1 ; i < = 12 ; i++)
            pPal- > palPalEntry[defaultOverride[i]] = defaultPalEntry[i];
        }

        m _ cPalette.CreatePalette(pPal);
        delete pPal;

        m _ pOldPalette = m _ pDC- > SelectPalette(&m _ cPalette, FALSE);
        m _ pDC- > RealizePalette();
    }
}
```

CreateRGBPalette()函数创建一个逻辑调色盘。为应用程序创建一个逻辑调色盘是很必要的。因为,你并不知道谁将运行你的应用程序,也不知道他在什么系统上运行程序。当创建了一个逻辑调色盘时,就应该检查所设置的像素格式,看一下系统是否有对建立逻辑调色盘的要求。如果结构 PIXELFORMATDESCRIPTOR 的成员 dwFlags 有 PFD _ NEED _ PALETTE 标志,则必须建立一个逻辑调色盘。这就是 CreateRGBPalette()函数为什么有开头的 6 条语句的原因。对于 CreateRGBPalette()函数的其他语句,我们在此不作解释。

(4) 编辑 Init()函数

在语句

```
::DescribePixelFormat(m _ pDC- > GetSafeHdc(), n, sizeof(pfd), &pfd);
```

之后添加:

```
CreateRGBPalette();
```

这样,通过 OnCreate()函数调用 Init()函数就创建了逻辑调色盘。此时,逻辑调色盘就作为一个 GDI 对象。

(5) 编辑 OnDestroy()函数

在语句 if (hrc) ::wglDeleteContext(hrc);之后添加:

```
if (m _ pOldPalette)
m _ pDC- > SelectPalette(m _ pOldPalette, FALSE);
```

此时,OnDestroy()函数有如下形式:

```
void CLightView::OnDestroy()
{
    HGLRC hrc;

    hrc = ::wglGetCurrentContext();

    ::wglMakeCurrent(NULL, NULL);
```

```
    if (hrc)
      ::wglDeleteContext(hrc);

    if (m_pOldPalette)
      m_pDC->SelectPalette(m_pOldPalette, FALSE);

    if (m_pDC)
      delete m_pDC;
    CView::OnDestroy();
}
```

(6) 响应调色消息

当一个应用程序被激活时,它可能实现自己的逻辑调色盘,同时意味着系统调色盘的颜色可能会被改变,从而可使留在背景上的其他的重叠窗口和弹出窗口看起来很奇怪。解决这一问题的办法就是要通知它们,系统调色盘已被改变。可以通过向它们发送 Windows 消息 WM_PALETTECANGED 和 WM_QUERYNEWPALETTE 来实现。

Windows 向即将激活的窗口发送 WM_QUERYNEWPALETTE 消息,当窗口对象接收到该消息时,它应该实现其逻辑调色盘,同时应使窗口用户区无效,并返回 TRUE 以通知 Windows 已改变了系统调色盘。

当活动窗口通过实现其逻辑调色盘而改变了系统调色盘时,Windows 向所有重叠窗口和弹出窗口发送 WM_PALETTECANGED 消息。下面我们实现这一过程。

打开 MainFrm.cpp 文件,为 CMainFrame 类添加 WM_QUERYNEWPALETTE 消息和 WM_PALETTECANGED 消息,并编辑如下:

```
BOOL CMainFrame::OnQueryNewPalette()
{
    // TODO: Add your message handler code here and/or call default
    WORD        i;
    CPalette    *pOldPal;
    CLightView  *pView = (CLightView *)GetActiveView();
    CClientDC   dc(pView);


    pOldPal = dc.SelectPalette(&pView->m_cPalette, FALSE);
    i = dc.RealizePalette();
    dc.SelectPalette(pOldPal, FALSE);

    if(i > 0)
        InvalidateRect(NULL);

    return CFrameWnd::OnQueryNewPalette();
}

void CMainFrame::OnPaletteChanged(CWnd * pFocusWnd)
```

```
{
    CFrameWnd::OnPaletteChanged(pFocusWnd);

    // TODO: Add your message handler code here
    CFrameWnd::OnPaletteChanged(pFocusWnd);

    if(pFocusWnd ! = this)
        OnQueryNewPalette();
}
```

(7) 文件包含

在 MainFrm.cpp 文件头部包含下列文件:

```
#include "LightDoc.h"
#include "LightView.h"
```

(8) 添加消息

打开 LightView.cpp 文件,添加 Windows 消息 WM_ERASEBKGND。删除已有语句:

```
return CView::OnEraseBkgnd(pDC);
```

只添加一条语句, 结果如下:

```
BOOL CLightView::OnEraseBkgnd(CDC * pDC)
{
    // TODO: Add your message handler code here and/or call default
    return TRUE;
}
```

(9) 运行程序

再次编译连接并运行程序。当然画面看起来没有变化。但从应用程序本身,从用户的角度讲,这是很有必要的。一个产品,如果不为用户着想,是没有竞争力的。

7.12 应用程序 Light 使用导示

我们已经生成了一个较完整的应用程序 Light,它是一个时实交互性的应用程序。光照、材质、颜色等参数实时地控制了物体最后的显示结果。为了更好地理解光照、材质等对物体颜色的影响,我们在此给出一些 Light 的使用帮助信息。

(1)聚光被关闭的情况

第一种光源的各个参数如下:

环境光: R=0.2,G=0.2,B=0.2

漫反射光: R=1.0,G=1.0,B=1.0

镜面反射光: R=1.0,G=1.0,B=1.0

位置: X=1.0,Y=1.0,Z=1.0。

可见,这束光的颜色为白色。由于程序内定,同时它是无限光源。

下面,我们只对材质属性进行改变:

1) 仅有漫反射光

环境色系数: R=0.0,G=0.0,B=0.0

扩散色系数: R=0.8,G=0.8,B=0.8

镜面高亮色系数: R=0.0,G=0.0,B=0.0

辐射光色系数: R=0.0,G=0.0,B=0.0

光度: Shininess=0.0

2) 有环境光和漫反射光

环境色系数: R=0.3,G=0.3,B=0.3

扩散色系数: R=0.8,G=0.8,B=0.8

镜面高亮色系数: R=0.0,G=0.0,B=0.0

辐射光色系数: R=0.0,G=0.0,B=0.0

光度: Shininess=0.0

3) 有漫反射光、镜面反射光和低高光

环境色系数: R=0.0,G=0.0,B=0.0

扩散色系数: R=0.8,G=0.8,B=0.8

镜面高亮色系数: R=1.0,G=1.0,B=1.0

辐射光色系数: R=0.0,G=0.0,B=0.0

光度: Shininess=10.0

4) 有漫反射光、镜面反射光和强高光

环境色系数: R=0.0,G=0.0,B=0.0

扩散色系数: R=0.8,G=0.8,B=0.8

镜面高亮色系数: R=1.0,G=1.0,B=1.0

辐射光色系数: R=0.0,G=0.0,B=0.0

光度: Shininess=100.0

5) 有漫反射光和镜面反射光

环境色系数: R=0.0,G=0.0,B=0.0

扩散色系数: R=0.8,G=0.8,B=0.8

镜面高亮色系数: R=1.0,G=1.0,B=1.0

辐射光色系数: R=0.0,G=0.0,B=0.0

光度: Shininess=0.0

6) 有漫反射光和辐射光

环境色系数: R=0.0,G=0.0,B=0.0

扩散色系数: R=0.8,G=0.8,B=0.8

镜面高亮色系数: R=0.0,G=0.0,B=0.0

辐射光色系数: R=0.4,G=0.4,B=0.4

光度: Shininess=0.0

7) 有漫反射光、镜面反射光、强高光和辐射光

环境色系数: R=0.0,G=0.0,B=0.0

扩散色系数: R=0.8,G=0.8,B=0.8

镜面高亮色系数：　　R = 1.0, G = 1.0, B = 1.0

辐射光色系数：　　R = 0.4, G = 0.4, B = 0.4

光度：　　Shininess = 100.0

8）有镜面反射光、强高光和辐射光，无漫反射光。

环境色系数：　　R = 0.0, G = 0.0, B = 0.0

扩散色系数：　　R = 0.0, G = 0.0, B = 0.0

镜面高亮色系数：　　R = 1.0, G = 1.0, B = 1.0

辐射光色系数：　　R = 0.4, G = 0.4, B = 0.4

光度：　　Shininess = 100.0

9）有黄色环境光、镜面反射光、强高光和辐射光，无漫反射光。

环境色系数：　　R = 0.8, G = 0.8, B = 0.8

扩散色系数：　　R = 0.0, G = 0.0, B = 0.0

镜面高亮色系数：　　R = 1.0, G = 1.0, B = 1.0

辐射光色系数：　　R = 0.4, G = 0.4, B = 0.4

光度：　　Shininess = 100.0

总之，还可以作出各种不同的变化。请自行对比参数和输出结果。

（2）聚光被打开的情况

光源属性采用程序提供的属性，分别进行常数、线性、二次衰减以及三者的组合操作。看看有什么不同之处。

（3）光源和材质打开的情况

在此情况下，执行红色、绿色、蓝色等颜色屏蔽操作，看看有什么情况发生，并比较其不同之处。

第 8 章　OpenGL 动画

OpenGL 支持双缓存(Double-Buffer)技术,该技术提供了一种生成平滑动画的机制。本章将学习怎样用 OpenGL 生成动画,而且是多么简单!

8.1　动画生成技术

计算机动画是指用程序或工具生成一系列的静态画面,然后通过画面的连续播放来反应对象的运动变化过程。动画是运动中的艺术。动画大师 John Halas 指出,运动是动画的要素。动画中的运动不仅指物体的运动,还包括视点的移动、视角的改变、光照的变化、纹理和色彩的变化等。

计算机动画技术最初是应影视业发展的要求而产生的。近几年来,由于科学技术的不断进步,计算机动画技术已成为综合运用计算机图形学、物理学、机械学、心理学、美学、生命科学及人工智能等学科或领域的知识来研究客观存在或高度抽象的物体的运动表现形式的一种高新技术,已经越来越成功地运用到科学研究、广告及影视制作等领域中。

按不同的动画生成技术可以将计算机动画分为:关键帧动画、变形动画、关节动画、人体动画和基于物理的动画。

关键帧的概念源于传统的卡通动画。在早期的卡通动画的制作过程中,熟练的动画设计师设计卡通片的关键画面,即关键帧。然后由一般的动画设计师设计中间帧。对于计算机动画,其中间帧是由计算机通过插值计算的方法来完成的。所有影响画面图像的参数都是关键帧的控制参数,如位置、纹理、光照等参数。样条驱动的动画主要用来表现结构的运动。在此方法中,用户交互地给出物体或结构的运动轨迹样条。无论是样条驱动的动画还是关键帧动画,都会遇到同一问题,即,给定运动轨迹,求物体在某一帧的位置。基于物理的动画有时也被称为运动动画,其运动对象要符合物理规律。一般运动动画(即通常所说的刚体运动动画)要符合运动学和动力学规律,这种运动动画比较容易实现。运动动画的另一个重要部分就是所谓的碰撞检测,即检查物体间是否发生了相互碰撞,根据场景中三维物体的形状,目前已有很多碰撞检测方法,这些方法包括半径法、包围盒法和标准平面方程法等。由于使用运动动画能比较贴切地反映现实世界,所以运动动画在科学研究和工程模拟、教育等领域是一个很有价值的工具。本书第 16 章将实现牛头刨床急回机构的运动仿真。

计算机动画中驱动要素是时间。计算机动画是通过足够快的速度显示一系列的单个帧以生成活动的感觉。一般来说,动画播放要到每秒 15 帧以上时才能连续,实际动画中设计的速度取决于记录动画的媒介。例如,电影业的标准是每秒 24 帧,欧洲 PAL 制式视频标准是每秒 25 帧,而美国的 NTSC 制式则用每秒 30 帧。

8.2　OpenGL 动画工作原理

OpenGL 是在帧缓存中存储和操作像素数据。帧缓存是由多个缓存组成的逻辑集,这些

缓存包括颜色缓存、深度缓存、累积缓存和模板缓存。OpenGL 是用双缓存技术来实现动画的。程序把帧存看成是两个视频缓存,在任意时刻,只有两者中的一个内容能被显示出来。当前可见视频缓存称为前台视频缓存(Front Buffer 或 On-Screen Buffer),不可见的正在画的视频缓存称为后台视频缓存(Back Buffer 或 Off-Screen Buffer)。当后台视频缓存中的内容被要求显示时,OpenGL 就会将它拷贝至前台视频缓存。显示硬件则不断地读可见视频缓存中的内容,并把结果显示在屏幕上。

双缓存技术可以生成平滑的动画,最好把一幅完全画好的图像显示一定的时间,然后提供下一幅画面,该画面也是完全画好的,并且显示同样长的时间。视频图像按此方式交替出现。从一幅图像变化到下一幅图像,由于时间极短,人眼是不会感觉到这种变化的。

在双缓存模式下,位平面被分成前台位平面和后台位平面。只有前台位平面被显示,因为绘制函数所绘制的场景首先被写向后台位平面。在画任何物体之前,必须把帧存设置成正确的组态。当完整的画面在后台视频缓存中画出以后,就调用 SwapBuffers()函数,使其成为可见视频缓存。

SwapBuffers()函数要等到下一次屏幕更新之后才能交换前台和后台视频缓存中的内容。图 8-1 表示该函数的工作过程。

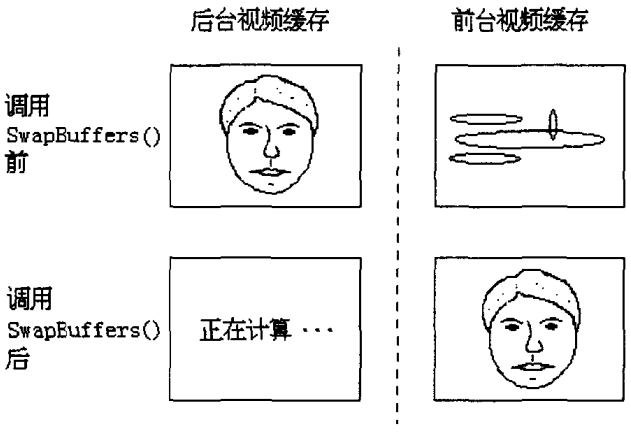

图 8-1 双缓存机制

必须注意,在交换前台和后台缓存中的内容之前,必须进行同步操作。OpenGL 提供了两个同步操作函数:glFlush()和 glFinish()。

glFlush()函数用于表明先前被送到 OpenGL 的所有指令必须在有限时间内完成。在网络模式下,该函数可以强制客户端在由命令组成的网络包还没有填满的情况下不等待命令而发送该包,以便绘制工作可以继续进行。而 glFinish()函数则用于强制完成所有先前的 OpenGL 指令。该命令要等到所有的绘制命令均得以完成之后才返回,在网络模式下,该函数是一个重要的同步操作函数。

8.3 让盒子动起来

为了简明地说明动画制作原理,本节我们的目标是让一个盒子转动起来。在第 16 章,我们还有更好的学习内容,即怎样建立时间和数据驱动的动画机制。

(1) 极小化 OpenGL 应用程序

建立一个工程名为 Animation 的工程,并用第 5 章中极小化 OpenGL 应用程序 GLTrans 的步骤极小化 Animation 工程。

(2) 编辑 bSetupPixelFormat()函数

将结构 PIXELFORMATDESCRIPTOR 中的内容更换为:

```
static PIXELFORMATDESCRIPTOR pfd =
{
  sizeof(PIXELFORMATDESCRIPTOR),
  1,
  PFD_DRAW_TO_WINDOW |
    PFD_SUPPORT_OPENGL |
    PFD_DOUBLEBUFFER,
  PFD_TYPE_RGBA,
  24,
  0, 0, 0, 0, 0, 0,
  0,0,0,0, 0, 0, 0,
  32,
    0,
    0,
  PFD_MAIN_PLANE,
  0,0, 0, 0
};
```

(3) 添加保护成员变量

添加以下保护成员变量:

```
GLfloat   m_fAngleY;
GLfloat   m_fAngleX;
GLfloat   m_fAngleZ;
BOOL      m_play;
```

初始化为:

```
m_fAngleY = 10.0f;
m_fAngleX = 1.0f;
m_fAngleZ = 5.0f;
m_play = FALSE;
```

变量 m_fAngleX,m_fAngleY,m_fAngleZ 分别说明盒子绕 x,y,z 轴旋转和角度。m_play 控制动画的播放,其值通过菜单来控制。

(4) 增加 WM_TIMER 消息

添加 WM_TIMER 消息响应函数 OnTimer(),并编辑如下:

```
void CAnimationView::OnTimer(UINT nIDEvent)
{
```

```
    // TODO: Add your message handler code here and/or call default

    m_fAngleY += 10.0f;
    m_fAngleX += 1.0f;
    m_fAngleZ += 5.0f;

    Invalidate(FALSE);

    CView::OnTimer(nIDEvent);
}
```

(5) 编辑 DrawScene()函数

```
void CAnimationView::DrawScene()
{
    static BOOL     bBusy = FALSE;

    if(bBusy)    return;
    bBusy = TRUE;

    glClearColor(1.0f,1.0f,1.0f,0.0f);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glTranslatef(0.0f,0.0f,-4.5f);

    glPushMatrix();

      glRotatef(m_fAngleX, 1.0f, 0.0f, 0.0f);
      glRotatef(m_fAngleY, 0.0f, 1.0f, 0.0f);
      glRotatef(m_fAngleZ, 0.0f, 0.0f, 1.0f);

        glBegin(GL_QUAD_STRIP);
            glColor3f(1.0f, 0.0f, 1.0f);
            glVertex3f(-0.5f, 0.5f, 0.5f);

            glColor3f(1.0f, 0.0f, 0.0f);
            glVertex3f(-0.5f, -0.5f, 0.5f);

            glColor3f(1.0f, 1.0f, 1.0f);
            glVertex3f(0.5f, 0.5f, 0.5f);

            glColor3f(1.0f, 1.0f, 0.0f);
            glVertex3f(0.5f, -0.5f, 0.5f);
```

```
        glColor3f(0.0f, 1.0f, 1.0f);
        glVertex3f(0.5f, 0.5f, -0.5f);

        glColor3f(0.0f, 1.0f, 0.0f);
        glVertex3f(0.5f, -0.5f, -0.5f);

        glColor3f(0.0f, 0.0f, 1.0f);
        glVertex3f(-0.5f, 0.5f, -0.5f);

        glColor3f(0.0f, 0.0f, 0.0f);
        glVertex3f(-0.5f, -0.5f, -0.5f);

        glColor3f(1.0f, 0.0f, 1.0f);
        glVertex3f(-0.5f, 0.5f, 0.5f);

        glColor3f(1.0f, 0.0f, 0.0f);
        glVertex3f(-0.5f, -0.5f, 0.5f);
glEnd();

        glBegin(GL_QUADS);
        glColor3f(1.0f, 0.0f, 1.0f);
        glVertex3f(-0.5f, 0.5f, 0.5f);

        glColor3f(1.0f, 1.0f, 1.0f);
        glVertex3f(0.5f, 0.5f, 0.5f);

        glColor3f(0.0f, 1.0f, 1.0f);
        glVertex3f(0.5f, 0.5f, -0.5f);

        glColor3f(0.0f, 0.0f, 1.0f);
        glVertex3f(-0.5f, 0.5f, -0.5f);
glEnd();

glBegin(GL_QUADS);
        glColor3f(1.0f, 0.0f, 0.0f);
        glVertex3f(-0.5f, -0.5f, 0.5f);

        glColor3f(1.0f, 1.0f, 0.0f);
        glVertex3f(0.5f, -0.5f, 0.5f);

        glColor3f(0.0f, 1.0f, 0.0f);
        glVertex3f(0.5f, -0.5f, -0.5f);
```

```
            glColor3f(0.0f, 0.0f, 0.0f);
            glVertex3f(-0.5f, -0.5f, -0.5f);
        glEnd();

    glPopMatrix();

    glFinish();
    SwapBuffers(wglGetCurrentDC());

    bBusy = FALSE;
}
```

函数 SwapBuffers()启动 OpenGL 的双缓存机制,实现后台视频缓存向前台视频缓存的图像传递。

(6) 编辑菜单

添加一个名为 Control 的菜单,即在属性对话框的 Caption 栏内加入 C&ontrol。增加一个菜单项,其 Caption 为 P&lay/Pause,其 ID 号为 IDM _ PALY _ PAUSE。对 IDM _ PALY _ PAUSE 消息进行响应:

```
void CAnimationView::OnPlayPause()
{
    // TODO: Add your command handler code here

    m_play = m_play ? FALSE : TRUE;
    if (m_play)
        SetTimer(1, 45, NULL);
    else
        KillTimer(1);
}

void CAnimationView::OnUpdatePlayPause(CCmdUI* pCmdUI)
{
    // TODO: Add your command update UI handler code here

}
```

在 Windows 环境下进行三维动画设计时,需要使用系统定时器。系统定时器作为 Windows 的一种输入设备,它可以周期性地提醒应用程序应该完成哪些工作。系统定时器可用于计时,唤醒应用程序,切换多任务,处理自动存储问题,保持恒定的速率和终止应用程序等。

Windows 为程序开发提供 SetTimer()函数来为应用程序分配一个系统定时器。通过该函数的参数可以设置时间间隔,Windows 将用这个时间间隔值周期性地向程序发送 WM _ TIMER 消息。可设置的时间间隔值一般在 1ms 至 65535ms 之间。比如:

```
SetTimer(1, 45, NULL);
```

设置序号为 1 的定时器,该定时器每 45ms 会发送一次 WM _ TIMER 消息。但是应该注意,SetTimer()函数设置的时间间隔最后将被处理成时钟片的整数倍。Windows 定时器和 PC 基本定时器一般具有相同的时间分辨率。一般地,Windows 发送给应用程序的 WM _ TIMER 消息并不是异步的。与其他消息一样,该消息也被放入消息队列之中。因此,在调用 Setimer()函数时,如果设置 1000ms 的时间间隔,并不能保证应用程序每间隔 1s(精确地说为 989ms)就收到一个 WM _ TIMER 消息。假设另一个应用程序因忙碌而超过该时间间隔,在这段时间内应用程序就有可能收不到任何 WM _ TIMER 消息。只有当忙碌的程序将控制返回给 Windows 后,应用程序才能从队列中获得消息。WM _ TIMER 和 WM _ PAINT 消息的优先级都较低。

在利用 OpenGL 设计动画时,时间间隔起着十分重要的作用。时间间隔很大时,将浪费很多系统时间,时间间隔较小时,视频不能进行正确地刷新,将有可能产生帧信息的丢失。在确定时间间隔时,一般应综合考虑系统的视频刷新频率和计算的复杂度等因素。对于网络程序,时间的控制将变得更加复杂。

如果需要在程序的整个存在期间使用定时器,可以在 WM _ CREATE 消息过程中设置定时器。比如:

```
int CAnimationView::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    if (CView::OnCreate(lpCreateStruct) == -1)
        return -1;

    // TODO: Add your specialized creation code here
    Init();

    SetTimer(1, 45, NULL);

    return 0;
}
```

当不再用定时器后,可用 KillTimer()函数取消定时器。

(7) 编辑函数 OnDestroy()

在其首部增加以下语句,使其作为该函数的第一条语句:

```
KillTimer(1);
```

(8) 运行程序

编译连接并运行程序,执行菜单,播放动画和暂停动画。图 8-2 是盒子在运行中的一个运行状态。

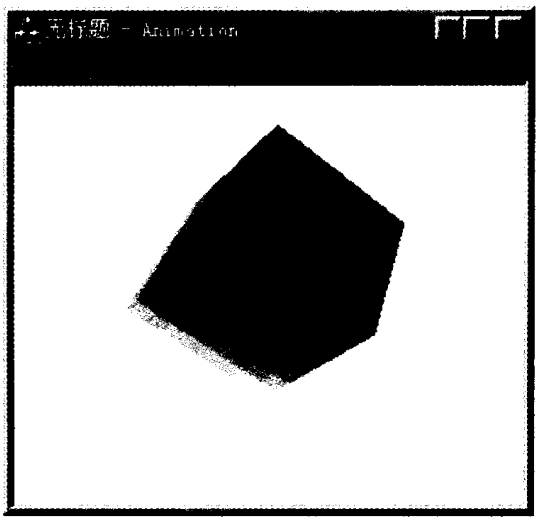

图 8-2　运动中的盒子

第 9 章　OpenGL 图像处理基础

在图像处理中，仅仅描述物体的几何结构是不够的，还需要反应物体的表面细节。比如，一张木质的桌子看起来就应该具有木纹等。这就需要特殊的方法进行表面处理。一般来说，这种方法就是对像素进行操作，将特定图像映射到物体的表面。

物体的几何顶点数据描述物体在空间的位置，而像素数据则用作图像的效果处理。本章将介绍：

- 怎样读写 OpenGL 图像；
- 怎样拷贝 OpenGL 图像；
- 怎样缩放 OpenGL 图像；
- 怎样生成位图字符和文本。

9.1　位图和图像

OpenGL 支持两种类型的图像对象，一种是单色的位图掩码，主要用来创建字符。另一种是彩色位图，它可以是一个图标或一幅真彩图像。由于不同计算机存储图像的方式不同，加之 OpenGL 是一个跨平台的开放系统，因此，OpenGL 须支持所有可能的图像数据格式。表 9-1 说明了 OpenGL 所支持的图像数据格式及其含义。

表 9-1

常　量	含　义
GL_COLOR_INDEX	单个颜色索引
GL_RGB	依次为红色、绿色、蓝色分量
GL_RGBA	依次为红、绿、蓝和 Alpha 分量
GL_RED	单个红色分量
GL_GREEN	单个绿色分量
GL_BLUE	单个蓝色分量
GL_ALPHA	单个 Alpha 分量
GL_LUMINANCE_ALPHA	亮度及分量
GL_LUMINANCE	单个亮度分量
GL_STENCIL_INDEX	单个模板索引
GL_DEPTH_COMPONENT	单个深度分量

一般地，数据的大小随着图像类型的不同而不同，同时也随着计算机类型而各异。因此，为了保证跨平台特性，OpenGL 又定义了一套图像数据类型。表 9-2 描述了这些数据类型和大小。

表 9-2

类　型	含　义
GL_BITMAP	无符号 8 位整型数组中的单个位
GL_BYTE	8 位整数
GL_UNSIGNED_BYTE	无符号的 8 位整数
GL_SHORT	16 位整数
GL_UNSIGNED_SHORT	无符号的 16 位整数
GL_INT	32 位整数
GL_UNSIGNED_INT	无符号的 32 位整数
GL_FLOAT	单精度浮点数

9.1.1 读取图像

有 3 个最基本的函数操作像素数据，它们是：glReadPixels()，glDrawPixels()及 glCopyPixels()。glReadPixels()函数从颜色缓存中读取一块矩阵像素并将其存储在内存中，该函数的原型为：

```
void glReadPixels( GLint x, GLint y, GLsizei width, GLsizei height, GLenum format,
                   GLenum type, GLvoid * pixels );
```

比如：

```
GLubyte image[64][64][3];
glReadPixels(0,0,64,64,GL_RGB,GL_UNSIGNED_BYTE,image);
```

在上述代码上，存储区域被定义为一个三维阵列，存储一个 64 x 64 位的图像，每位包含像素的 3 个分量信息，即 R、G、B 值。glReadPixels()函数则从颜色缓存中读取数据，并将其存储在 image 数组中。该函数共有 7 个参数。此时，图像区域的左下角点坐标为(0,0)，图像的宽度和高度均为 64，像素数据的格式为 GL_RGB，像素数据的类型为 GL_UNSIGNED_BYTE，最后一个参数是存储地址。

9.1.2 绘制图像

一旦将图像数据读入内存，就可以用 glDrawPixels()函数将其显示在屏幕上。该函数的原型为：

```
void glDrawPixels( GLsizei width, GLsizei height, GLenum format, GLenum type,
                   const GLvoid * pixels );
```

相对应地，前面已读入的图像可用下述语句来显示：

```
glDrawPixels(64,64,GL_RGB,GL_UNSIGNED_BYTE,image);
```

glDrawPixels()函数将图像显示在当前光栅位置，共有 5 个参数：图像的宽度和高度，图像像素

数据格式和类型,图像地址。

9.1.3 当前光栅位置

OpenGL不仅可以输出图像,而且还可以控制图像显示的位置。图像显示的位置是与当前光栅位置相关的。设置当前光栅位置的函数是glRasterPos()。该函数共有24种不同的形式。比如:

```
glRasterPos3f(0.6f,0.5f,0.0f);
```

glRasterPos3f()函数设置当前所画位图或图像的原点。其中,光栅位置为(0.6f,0.5f,0.0f),这就是所画位图或图像的原点位置。要注意,这个坐标不同于函数所使用的屏幕坐标,要受到模型变换和投影变换的影响。

9.1.4 拷贝图像

第三个最基本的OpenGL图像处理函数为glCopyPixels(),可以拷贝一个图像至另一个区域,其原型为:

```
void glCopyPixels( GLint x, GLint y, GLsizei width, GLsizei height, GLenum type );
```

该函数不需要将图像读至一个缓存,而是在帧存中直接读取并写向帧存。它共有5个参数。glCopyPixels()函数拷贝帧存中左下角点坐标为(x,y),宽度和高度分别为width和height的一块矩形区域中的像素数据。所获得的数据则写向新的位置,其左下角点在当前光栅位置。参数type可以为GL_COLOR、GL_DEPTH和GL_STENCIL。若type为GL_COLOR,像素数据的格式为GL_RGB或GL_COLOR_INDEX;若type为GL_DEPTH,像素数据的格式为GL_DEPTH_COMPONENT;若type为GL_STENCIL,像素数据的格式则为GL_STENCIL_INDEX。

9.1.5 缩放图像

OpenGL可以对像素进行缩放。该函数为glPixelZoom(),其原型为:

```
void glPixelZoom( GLfloat xfactor, GLfloat yfactor );
```

glPixelZoom()函数设置像素写操作沿x和y方向的缩放因子。x,y可以为负值。

与像素操作有关的函数还有:说明像素存储模式的函数glPixelStore(),传递像素的函数glPixelTransfer()和映射像素的函数glPixelMap()。这3个函数的使用我们在此不作介绍,本书下一章将用到glPixelStore()函数,而glPixelTransfer()及glPixelMap()函数本书并不涉及,请使用在线帮助。不过,像素操作是高级图像处理的基础。

9.1.6 图像操作工程Image

下面建立一个工程,读者可从中学习如何进行图像操作。

(1) 建立工程

建立一个名为 Image 的工程,并按第 5 章中的方法极小化 OpenGL。

(2) 增加一个保护成员函数

添加一个名为 myImage 的保护成员函数,并编辑如下:

```
void CImageView::myImage()
{
    GLubyte image1[64][64][3];
    GLubyte image2[64][64][3];
    for(int i=0;i<64;i++)
        for(int j=0;j<64;j++)
        {
            image1[i][j][0]=(GLubyte)j*3.5;
            image1[i][j][1]=(GLubyte)i*3.5;
            image1[i][j][2]=(GLubyte)(255-i*3.5);
        }
    //画至左上角:
    glRasterPos3f(-1.2f,0.5f,0.0f);
    glDrawPixels(64,64,GL_RGB,GL_UNSIGNED_BYTE,image1);
    GLint rasPostion[4];
    glGetIntegerv(GL_CURRENT_RASTER_POSITION,rasPostion);
    glReadPixels(rasPostion[0],rasPostion[1],64,64,GL_RGB,GL_UNSIGNED_BYTE,
                 image2);
    //画至右上角:
    glRasterPos3f(0.6f,0.5f,0.0f);
    glDrawPixels(64,64,GL_RGB,GL_UNSIGNED_BYTE,image2);
    glGetIntegerv(GL_CURRENT_RASTER_POSITION,rasPostion);
    //取一半,画至左下角:
    glRasterPos3f(-1.2f,-0.3f,0.0f);
    glCopyPixels(rasPostion[0],rasPostion[1],64,32,GL_COLOR);
    //取四分之一,宽度放大一倍,并画至右下角:
    glRasterPos3f(0.6f,-0.3f,0.0f);
    glPixelZoom(2.0f,1.0f);
    glCopyPixels(rasPostion[0],rasPostion[1],32,32,GL_COLOR);
}
```

(3) 编辑 DrawScene()函数

```
void CImageView::DrawScene()
{
    glClearColor(1.0f,1.0f,1.0f,1.0f);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glTranslatef(0.0f,0.0f,-4.5f);
```

```
    myImage();
    glFinish();
}
```

(4) 运行程序

编译连接并运行程序,其结果如图 9-1 所示。

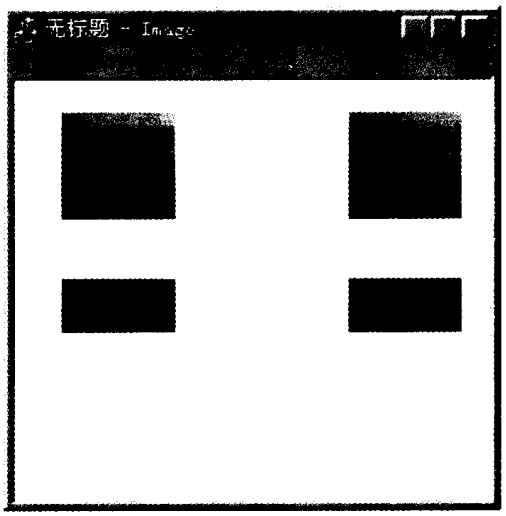

图 9-1 图像处理

现在,如果给定一个位图文件,通过适当的处理并按上述方法就可以将位图显示出来。

9.2 位图字符和文本

OpenGL 可以直接处理单色位图。单色位图的存储形式比较简单,如果存储位为 1 则用指定的颜色填充该位,如果存储位为 0,则用底色填充该位。如果按每行 8 位扫描,则按以下数组表示的位所生成的位图则是一个字母 B,如图 9-2 示。

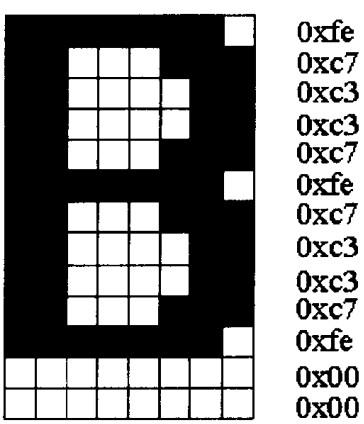

图 9-2 字符 B

```
GLubyte letters[13] = {
    0x00,  0x00,  0xfe,  0xc7,  0xc3,  0xc3,  0xc7,
    0xfe,  0xc7,  0xc3,  0xc3,  0xc7,  0xfe  };
```

要输出位图,还需要调 glBitmap()函数把位图按指定的位置显示在屏幕上,其原型为:

```
void glBitmap( GLsizei width,GLsizei height, GLfloat xorig, GLfloat yorig, GLfloat xmove,
                GLfloat ymove, const GLubyte * bitmap );
```

其中,width 指定位图的宽,height 指定位图的高,均以像素为单位;(xorig,yorig)为位图原点的坐标;(xmove,ymove)为位图基于当前光栅位置的偏移量;bitmap 则为存储位图的地址。

9.2.1 生成位图字符

请按以下步骤操作:

(1) 添加一个保护成员函数

添加一个保护成员函数 BitmapFont(),并编辑如下:

```
void CImageView::BitmapFont()
{
    GLubyte letters[][13] = {
        { 0x00, 0x00, 0xc3, 0xc3, 0xc3, 0xc3, 0xff, 0xc3, 0xc3, 0xc3, 0x66, 0x3c, 0x18},
        { 0x00, 0x00, 0xfe, 0xc7, 0xc3, 0xc3, 0xc7,0xfe, 0xc7, 0xc3, 0xc3, 0xc7, 0xfe },
        { 0x00, 0x00, 0x7e, 0xe7, 0xc0, 0xc0, 0xc0,0xc0, 0xc0, 0xc0, 0xc0, 0xe7, 0x7e},
        { 0x00, 0x00, 0xfc, 0xce, 0xc7, 0xc3, 0xc3,0xc3, 0xc3, 0xc3, 0xc7, 0xce, 0xfc },
        { 0x00, 0x00, 0xff, 0xc0, 0xc0, 0xc0, 0xc0,0xfc, 0xc0, 0xc0, 0xc0, 0xc0, 0xff },
        { 0x00, 0x00, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0,0xc0, 0xfc, 0xc0, 0xc0, 0xc0, 0xff },
        { 0x00, 0x00, 0x7e, 0xe7, 0xc3, 0xc3, 0xcf, 0xc0, 0xc0, 0xc0, 0xc0, 0xe7, 0x7e},
        { 0x00, 0x00, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3,0xff, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3},
        { 0x00, 0x00, 0x7e, 0x18, 0x18, 0x18, 0x18,0x18, 0x18, 0x18, 0x18, 0x18, 0x7e},
        { 0x00, 0x00, 0x7c, 0xee, 0xc6, 0x06, 0x06,0x06, 0x06, 0x06, 0x06, 0x06, 0x06},
        { 0x00, 0x00, 0xc3, 0xc6, 0xcc, 0xd8, 0xf0, 0xe0, 0xf0, 0xd8, 0xcc, 0xc6, 0xc3},
        { 0x00, 0x00, 0xff, 0xc0, 0xc0, 0xc0, 0xc0,0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0},
        { 0x00, 0x00, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3,0xc3, 0xdb, 0xff, 0xff, 0xe7, 0xc3},
        { 0x00, 0x00, 0xc7, 0xc7, 0xcf, 0xcf, 0xdf,0xdb, 0xfb, 0xf3, 0xf3, 0xe3, 0xe3},
        { 0x00, 0x00, 0x7e, 0xe7, 0xc3, 0xc3, 0xc3,0xc3, 0xc3, 0xc3, 0xc3, 0xe7, 0x7e},
        { 0x00, 0x00, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0,0xfe, 0xc7, 0xc3, 0xc3, 0xc7, 0xfe },
        { 0x00, 0x00, 0x3f, 0x6e, 0xdf, 0xdb, 0xc3,0xc3, 0xc3, 0xc3, 0xc3, 0x66, 0x3c},
        { 0x00, 0x00, 0xc3, 0xc6, 0xcc, 0xd8, 0xf0,0xfe, 0xc7, 0xc3, 0xc3, 0xc7, 0xfe },
        { 0x00, 0x00, 0x7e, 0xe7, 0x03, 0x03, 0x07,0x7e, 0xe0, 0xc0, 0xc0, 0xe7, 0x7e},
        { 0x00, 0x00, 0x18, 0x18, 0x18, 0x18, 0x18,0x18, 0x18, 0x18, 0x18, 0x18, 0xff },
        { 0x00, 0x00, 0x7e, 0xe7, 0xc3, 0xc3, 0xc3,0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3},
        { 0x00, 0x00, 0x18, 0x3c, 0x3c, 0x66, 0x66,0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3},
        { 0x00, 0x00, 0xc3, 0xe7, 0xff, 0xff, 0xdb,0xdb, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3},
        { 0x00, 0x00, 0xc3, 0x66, 0x66, 0x3c, 0x3c,0x18, 0x3c, 0x3c, 0x66, 0x66, 0xc3},
```

```
        { 0x00, 0x00, 0x18, 0x18, 0x18, 0x18, 0x18,0x18, 0x3c, 0x3c, 0x66, 0x66, 0xc3 },
        { 0x00, 0x00, 0xff, 0xc0, 0xc0, 0x60, 0x30,0x7e, 0x0c, 0x06, 0x03, 0x03, 0xff }
    };
    GLuint i, j;
    glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
    glColor3f(0.0f,0.0f,1.0f);
    glRasterPos3f(-1.2f, -0.7f,0.0f);
    for (i = 0; i < 26; i++)
        glBitmap(8, 13, 0.0, 0.0, 20.0, 0.0, letters[i]);
}
```

其中,glPixelStorei(GL_UNPACK_ALIGNMENT, 1)描述位图数据的存储方式。

(2) 编辑 DrawScene()函数

在语句 myImage();之后添加:

```
BitmapFont();
```

(3) 运行程序

再次编译连接并运行程序,程序除绘制 4 个图块外,再在其后部绘制从 A 到 Z 共 26 个字母。

要通过位图显示字符,必须首先生成表示位图的各个位,显然这是十分麻烦的,尤其是生成汉字字符。OpenGL 可以用 Windows 已有字库中的字体,生成文本。

9.2.2 生成文本

Microsoft Windows 95/ NT 所支持的 OpenGL 只能在单缓存模式下支持 GDI 字体和文本输出函数。本节用 Windows 已有字库中的字体,采用显示列表的方法来制作文本。

(1) 添加一个保护成员函数

添加一个保护成员函数 DrawText(),并编辑如下:

```
void CImageView::DrawText()
{
    wglUseFontBitmaps(wglGetCurrentDC(),0,256,1000);
    glListBase(1000);
    glRasterPos3f(-1.2f, -1.1f,0.0F);
    glCallLists(16,GL_UNSIGNED_BYTE,"HUST HuBei China");
}
```

上段代码采用显示列表来管理文本。wglUseFontBitmaps()函数创建一系列基于当前着色描述表的位图字符的显示列表。glListBase()函数为 glCallLists()函数设置显示列表基数。glCallLists()函数则调用显示列表并输出字符串。

(2) 编辑 DrawScene()函数

在语句 BitmapFont();之后添加:

```
DrawText();
```

(3) 运行程序

编译连接并运行程序,其结果是在画面底部添加一串文本,如图 9-3 所示。

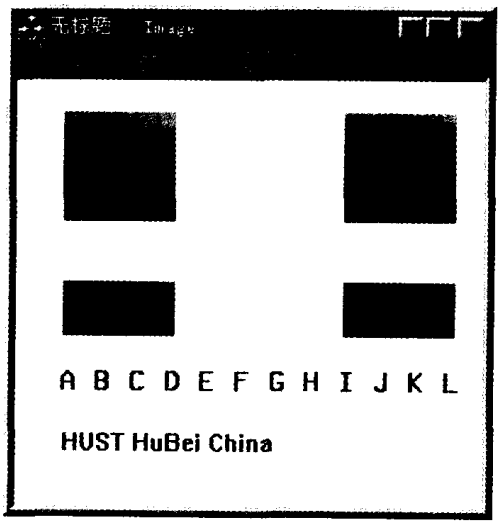

图 9-3 生成文本

第 10 章　纹理贴图

到目前为止，我们可以应用 OpenGL 生成各种物体，并可以赋予材质和对物体进行光照。但是，物体的表面过于光滑和单调，看起来反而不真实。这是因为现实世界中的物体其表面往往有各种纹理，即表面细节。例如，刨光的木材表面有木纹，建筑物墙壁上有装饰图案，机器外壳表面有文字说明它的名称、型号等。在计算机图像处理中需要用纹理贴图技术来解决这些问题。本章将介绍：

- 如何自定义纹理，并将纹理贴在物体上；
- 如何将 24 位真彩位图贴到物体上；
- 如何生成自动纹理坐标，并利用它们将一个位图贴到物体上；
- 如何进行 Mip 贴图。

10.1　纹理贴图技术

纹理贴图(Texture Mapping)技术也叫纹理映射技术。简单地说，纹理分为两种，通过颜色色彩或明暗的变化体现出来的表面细节，这种纹理称为颜色纹理；另一类纹理则是由于不规则的细小凹凸造成的，例如，桔子皮的皱纹和未磨光的凹痕等。生成颜色纹理的一般方法是在一平面区域(即纹理空间)上预先定义纹理图案，然后建立物体表面的点与纹理空间的点之间的对应(即映射)。当物体表面的可见点确定之后，以纹理空间的对应点的值乘以亮度值，就可以把纹理图案附到物体的表面上。也可以用类似的方法给物体表面产生凸凹不平的凸包纹理。不过这时纹理值作用在法线向量上，而不是作用于颜色亮度。纹理的定义有连续法和离散法两种。连续法把纹理定义为一个二元函数，函数的定义域就是纹理空间。离散法把纹理函数定义在一个二维数组中，代表纹理空间中行间隔和列间隔固定的一组网格点上的纹理值。网格点之间的其他点的纹理值则通过网格点的值插值获得。通过纹理空间与物体空间之间的坐标变换，把纹理贴图到物体表面。

纹理贴图类型主要有材质贴图(Texture Mapping)、Mip 贴图(Mip Mapping)、凹凸贴图(Bump Mapping)、动态材质贴图(Video Texture Mapping)等。材质贴图是将一张已有的图像(通常是位图)直接贴到物体的表面。Mip 贴图依据不同的精度要求，使用不同版本的材质图样进行贴图。凹凸贴图是一种在三维场景中模拟粗糙表面的一种贴图技术。它将深度的变化保存在一张贴图中，然后将对三维物体进行混合贴图处理，最后生成具有凹凸感的贴图效果。动态材质贴图技术可产生较好的材质贴图效果。它将一段连续的图像以材质的方式处理，然后贴到物体的表面上去。例如，在赛车游戏中，在挡风玻璃上贴一段连续的天空动画，就能产生类似即时反射环境贴图的效果。

随着计算机技术的发展和用户对高精度图片的要求，传统的纹理贴图技术已经得到了很大的发展，许多新技术被开发出来。目前应用较广泛的是双线性过滤(Bilinear Interpolation)技术。当材质(通常为一个位图图片)被贴到屏幕所显示的一个模型时，材质处理器必须决定每个图素要贴到的像素位置。由于材质是 2D 图片，而模型是 3D 物体，所以图素的范围与像

素范围通常不能很好地吻合。此时,为解决像素的贴图问题,就需要用插值的方法来解决。解决这一问题,早期采用近邻取样(Nearest Neighbor)技术。但近邻取样技术所获得的贴图品质较差,容易出现块状图。双线性过滤技术就被开发出来。双线性过滤技术先确定最接近像素的4个图素,然后对它们进行差补计算,其结果决定该点的颜色。该技术能减少纹理缩放时的块状图。

但是双线性过滤技术还不能解决纹理贴图的景深锯齿(Depth Alias)问题。景深锯齿现象是当一个物体远离视点时产生的,当一个物体正在远离视点时,贴在越来越小的物体上的图案也会被逐渐压缩,以致最后产生视觉余像,这些视觉余像在动画中会引起闪烁现象。为了消除景深锯齿使图像更符合实际情况,即远处物体的细节比近处物体的细节要少,图像工作者开发了一个新的技术,即多重贴图(Mip Mapping)技术,也就是我们经常说的 Mip 贴图。Mip 贴图是采用一系列的经过处理的不同大小的图像,当视线移近物体时,就以较高细节和较高清晰度的图像显示,反之就以较低细节和较低模糊度的图像显示,从而提高图像处理的整体效率。不过虽然 Mip 贴图解决了景深锯齿现象,但又产生了一个视觉问题,这就是当一个物体向远处移动且进行图像切换时,会出现闪烁现象。该现象在动画中表面十分突出且易于被人眼捕获。

三线性过滤(Trilinear Interpolation)技术就是用来解决这一问题的。通过采用这种技术,在图层切换时可以对贴图的底层物体进行平滑着色处理,因而能够解决大部分景深锯齿现象和闪烁问题。三线性过滤技术比传统的双线性过滤技术能够更明显地增强图形质量。通过去掉运动物体的余像使图形更平滑。三线性过滤技术要采用很多材质影像,而每张的大小恰好是前一张的四分之一。比如,若第一张材质影像大小为 512×512,第二张就会为 256×256,最后一张为 1×1。目前软件开发商正在寻找将三线性过滤技术和多重贴图相结合的新技术,即三线性多重贴图技术。

在贴图时,贴在物体上的图往往出现倾斜或错位,这就需要用透视校正(Perspective Correction)来处理。透视校正技术采用数学运算的方式,以确定贴在物体上的部分影像向透视消失的方向正确地收敛。这项技术可以减少直线的弯曲变形,增强贴图的真实效果。

为了模拟现实生活中的场景,就需要数量巨大的纹理细节,对内存的要求较大,因而纹理压缩技术是必要的。比如,多数图形加速卡的引擎使系统从主存中读取纹理数据时,增加了数据存储量。但是,主内存接口是共享资源,除了纹理数据以外,还被用于传递几何数据,同时系统内存还要被操作系统和其他程序所使用。因此,纹理数据的读取不能占用所有的主内存带宽。纹理压缩技术就是要解决纹理数据量与带宽所引起的问题。如 S3 公司的 S3TC 纹理压缩技术,微软已把它作为 DirectX 6.0 中的标准压缩技术。

纹理贴图技术是计算机图形学中不断研究的课题,有其自身的难度和现实的市场需要性。

10.2 OpenGL 纹理贴图

纹理贴图的数学过程是十分复杂的,但 OpenGL 已对其进行了处理,使编程者可以利用几个函数就可以完成纹理贴图。执行纹理贴图的步骤可以概括为:定义纹理贴图,控制纹理,说明纹理贴图方式,定义纹理坐标等。

10.2.1 定义纹理

这里主要采用离散法定义纹理。纹理可以是一维的,也可以是二维的。在图像处理中,通常采用的是二维纹理贴图。定义二维纹理贴图的函数是 glTexImage2D(),其原型为:

```
void glTexImage2D( GLenum target, GLint level, GLint components, GLsizei width,
    GLsizei height, GLint border, GLenum format, GLenum type, const GLvoid * pixels );
```

其中,target 参数必须是 GL_TEXTURE_2D。level 参数表示纹理图像纹理分辨率的级数。当只有一种分辨率时,level 为 0。components 参数可以为 1,2,3,4。1 代表只选择纹理图像的红色分量,2 代表选择纹理图像的红色和 Alpha 分量,3 代表同时选择纹理图像的红色、绿色和蓝色分量,4 代表选择其红、绿、蓝及 Alpha 分量。width 和 height 参数定义纹理图像的长度和宽度。border 参数则说明纹理边界的宽度。比如,当 border 为 0 时,边界也为 0;当 border 为 1 时,纹理图像的长度和宽度必须可以写成 $2m+2\times border$ 及 $2n+2\times border$ 的形式,其中 m,n 为一个大于 32 的整数。format 参数说明了纹理图像数据的格式,它们可以是 GL_COLOR_INDEX、GL_RED、GL_GREEN、GL_BLUE、GL_ALPHA、GL_RGB、GL_RGBA、GL_BGR_EXT、GL_BGRA_EXT、GL_LUMINANCE 或 GL_LUMINANCE_ALPHA。type 参数指明纹理图像数据类型,它们可以是 GL_UNSIGNED_BYTE、GL_BYTE、GL_BITMAP、GL_UNSIGNED_SHORT、GL_SHORT、GL_UNSIGNED_INT、GL_INT 或 GL_FLOAT。pixels 参数说明了纹理图像的像素(或纹素)数据的存储地址。通常地,纹素数据可以是色彩数据或亮度数据等。

定义一维纹理贴图的函数,其原型为:

```
void glTexImage1D( GLenum target, GLint level, GLint components, GLsizei width,
        GLsizei height, GLint border, GLenum format, GLenum type, const GLvoid *
pixels );
```

各参数的含义可参考 glTexImage2D()函数。

10.2.2 控制纹理

纹理图像在映射到物体时会产生许多问题。这些问题主要有纹理图像的纹理怎样对应到屏幕上的像素;当纹理图像较大或较小时,怎样通过纹理贴图实现纹理缩限和纹理重复等。OpenGL 通过 glTexParameter() 函数来解决这些问题。该函数的原型(以 glTexParameterf 形式为例)为:

```
void glTexParameterf( GLenum target, GLenum pname, GLfloat param );
```

其中 target 参数可以为 GL_TEXTURE_1D 和 GL_TEXTURE_2D。pname 参数的取值参见表 10-1。

表 10-1

常 量	含 义
GL_TEXTURE_MIN_FILTER	一个像素对应多个纹素
GL_TEXTURE_MAG_FILTER	多个像素对应一个纹素
GL_TEXTURE_WRAP_S	设置纹理在 s 方向上的被控行为
GL_TEXTURE_WRAP_T	设置纹理在 t 方向上的被控行为
GL_TEXTURE_BORDER_COLOR	设置边界颜色

param 参数的取值要根据 pname 而定,其可能取值为 GL_CLAMP、GL_REPEAT、GL_NEAREST、GL_LINEAR 、GL_NEAREST_MIPMAP_NEAREST、GL_LINEAR_MIPMAP_NEAREST 、GL_NEAREST_MIPMAP_LINEAR 或 GL_LINEAR_MIPMAP_LINEAR 。这些常量的含义我们在此不给出,可以参考 Visual C++ 在线帮助。该函数共有 4 种不同的形式。

10.2.3 说明纹理贴图方式

在进行纹理贴图时,纹理图像可以直接贴到物体上从而覆盖物体已有的颜色,也可以和物体的原有颜色进行融合。融合是一种重要的图像处理技术,主要用于透明效果处理(见下一章)。OpenGL 用于纹理贴图方式的函数为 glTexEnv()。该函数的原型(以 glTexEnvf 形式为例)为:

```
void glTexEnvf( GLenum target, GLenum pname, GLfloat param );
```

其中,target 参数必须是 GL_TEXTURE_ENV。其中 pname 参数可以取值 GL_TEXTURE_ENV_MODE 或 GL_TEXTURE_ENV_COLOR。而参数 param 为 GL_MODULATE、GL_DECAL 或 GL_BLEND。该函数共有 4 种不同的形式,其他形式的参数取值请参考 Visual C++ 在线帮助。

10.2.4 定义纹理坐标

在 OpenGL 中,许多信息是通过顶点来传递的。顶点信息有几何坐标、顶点法向量、顶点颜色等等。与顶点有关的另一类重要的图像控制信息是顶点的纹理坐标。纹理坐标控制纹理图像中的纹素怎样映射到物体。纹理坐标可以是 1、2、3、4 维的,通常也用齐次坐标来表示,即(s,t,r,q)。OpenGL 定义纹理坐标的函数为 glTexCoord()。该函数共有 32 种不同的形式。比如:

```
glTexCoord2f(0.0f,0.0f);
```

将纹理贴到矩形上时,计算纹理坐标是一件很容易的事,但是如果要将纹理贴到曲面上,就必须仔细并正确地计算纹理坐标。纹理坐标、纹理矩形大小以及被贴物体的表面形状将直接影响纹理贴图的效果。

10.2.5 应用纹理贴图技术

现在,我们来具体地应用纹理贴图技术进行编程练习。

(1) 建立工程

建立一个工程名为 Texture 的工程,并用第 5 章中极小化 OpenGL 应用程序 GLTrans 的步骤极小化 Texture 工程。

(2) 添加枚举变量及保护成员变量

打开 TextrueView.h 头文件,在其首部添加:

```
enum{Texture,DibTexture,AutoTexture, Mipmap};
```

添加保护成员变量如下:

```
GLint     m_typeControl;
```

初始化为:

```
m_typeControl = Texture;
```

(3) 添加保护成员函数

添加如下保护成员函数:TexturePlane(),并编辑如下:

```
void CTextureView::TexturePlane()
{
    GLubyte     image[64][64][3];

    //    创建纹理
    int i,j,g,b,c;
    for(i=0;i<64;i++)
        for(j=0;j<64;j++)
        {
            c=((((i & 0x04)==0)^((j & 0x04))==0))*255;
            g=(4*i)%255;
            b=(j*i)%255;
            image[i][j][0]=(GLubyte)c;
            image[i][j][1]=(GLubyte)g;
            image[i][j][2]=(GLubyte)b;
        }

        glPixelStorei(GL_UNPACK_ALIGNMENT,1);

    //    定义纹理:
    glTexImage2D(GL_TEXTURE_2D,0,3,64,64,
        0,GL_RGB,GL_UNSIGNED_BYTE,image);
```

```
//    控制纹理:
glTexParameterf(GL_TEXTURE_2D,GL_TEXTURE_WRAP_S,GL_CLAMP);
glTexParameterf(GL_TEXTURE_2D,GL_TEXTURE_WRAP_T,GL_CLAMP);
glTexParameterf(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,
                GL_NEAREST);
glTexParameterf(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,
                GL_NEAREST);

//    说明纹理贴图方式:
glTexEnvf(GL_TEXTURE_ENV,GL_TEXTURE_ENV_MODE,GL_DECAL);

//    启动纹理贴图:
glEnable(GL_TEXTURE_2D);
glShadeModel(GL_SMOOTH);

//    定义纹理坐标和物体几何坐标:
glBegin(GL_QUADS);
    glTexCoord2f(0.0f,0.0f);glVertex3f(-1.0f,-1.0f,0.0f);
    glTexCoord2f(0.0f,1.0f);glVertex3f(-1.0f,1.0f,0.0f);
    glTexCoord2f(1.0f,1.0f);glVertex3f(1.0f,1.0f,0.0f);
    glTexCoord2f(1.0f,0.0f);glVertex3f(1.0f,-1.0f,0.0f);
glEnd();

glDisable(GL_TEXTURE_2D);
}
```

注意,纹理贴图技术作为一种特殊的技术,在应用之前也必须进行启动:

```
glEnable(GL_TEXTURE_2D);
```

在程序不再需要进行纹理贴图技术时就应该将它关闭:

```
glDisable(GL_TEXTURE_2D);
```

(4)添加菜单

添加一个名为 Type 的菜单,即在属性对话框的 Caption 栏内加入 &Type。增加以下菜单项,表 10-2 列出了它们的属性对话框中的 Caption 及 ID 号。

表 10-2

Caption	ID
T&exture	IDM_TEXTURE
&Bitmap Texture	IDM_DIB_TEXTURE
&AutoTexture	IDM_AUTO_TEXTURE
&Mip Mapping	IDM_MIPMAPPING

并分别响应：

```
void CTextureView::OnTexture()
{
    // TODO: Add your command handler code here
    m_typeControl = Texture;
    Invalidate();
}

void CTextureView::OnUpdateTexture(CCmdUI* pCmdUI)
{
    // TODO: Add your command update UI handler code here
    if(m_typeControl == Texture)
        pCmdUI->SetCheck(TRUE);
    else
        pCmdUI->SetCheck(FALSE);
}

void CTextureView::OnDibTexture()
{
    // TODO: Add your command handler code here
    m_typeControl = DibTexture;
    Invalidate();
}

void CTextureView::OnUpdateDibTexture(CCmdUI* pCmdUI)
{
    // TODO: Add your command update UI handler code here
    if(m_typeControl == DibTexture)
        pCmdUI->SetCheck(TRUE);
    else
        pCmdUI->SetCheck(FALSE);
}

void CTextureView::OnAutoTexture()
{
    // TODO: Add your command handler code here
    m_typeControl = AutoTexture;
    Invalidate();
}

void CTextureView::OnUpdateAutoTexture(CCmdUI* pCmdUI)
{
    // TODO: Add your command update UI handler code here
```

```
    if(m_typeControl == AutoTexture)
        pCmdUI->SetCheck(TRUE);
    else
        pCmdUI->SetCheck(FALSE);
}

void CTextureView::OnMipmapping()
{
    // TODO: Add your command handler code here
    m_typeControl = Mipmap;
    Invalidate();
}

void CTextureView::OnUpdateMipmapping(CCmdUI* pCmdUI)
{
    // TODO: Add your command update UI handler code here
    if(m_typeControl == Mipmap)
        pCmdUI->SetCheck(TRUE);
    else
        pCmdUI->SetCheck(FALSE);
}
```

(5) 编辑函数 DrawScene()

```
void CTextureView::DrawScene()
{
    glClearColor(1.0f,1.0f,1.0f,1.0f);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glColor3f(1.0f, 0.0f, 0.0f);

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glTranslatef(0.0f,0.0f,-4.5f);

    switch(m_typeControl)
    {
        case Texture:       TexturePlane(); break;
    }

    glFinish();
}
```

(6) 运行程序

编译连接并运行程序,执行菜单 Type \ Texture(此时可以不执行此菜单,因为这是缺省设置),其运行结果如图 10-1 所示。

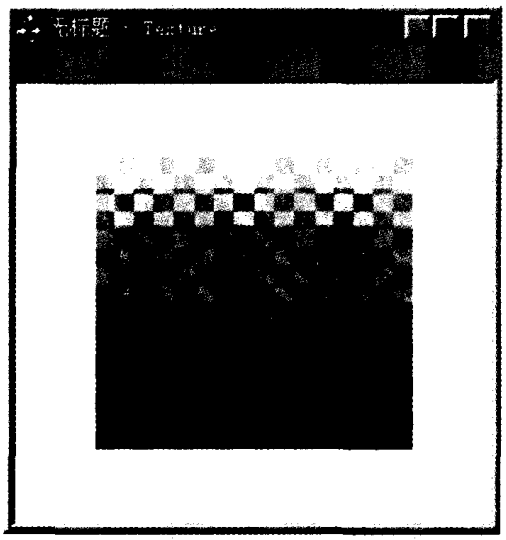

图 10-1　纹理贴图

10.3　位图纹理

本节的目标是将一个给定的位图贴到物体表面上。在 Windows 编程和图像处理中，位图是我们经常要处理的图形对象，能将位图显示在指定的客户区是 Windows 编程的基本要求。位图不仅能给应用程序带来丰富多彩的内容，而且在图像处理中占有十分重要的地位。位图可以作为图像合成的背景，也可以作为一种纹理贴在物体的表面。比如，为了制作出木质的桌子，可以先制作桌子的几何形体，接着制作一小块木纹，最后将木纹贴到桌子上。这是一种解决问题的方法，其关键是制作一块木纹，即生成一幅表示木纹图案的位图。值得庆贺的是已有许多图形图像制作软件可以完成这项工作，除 Windows 画笔外，还有 CorelDraw、PhotoShop、3DS MAX 等等。

10.3.1　设备无关位图(DIB)的数据结构

位图分为设备相关位图和设备无关位图。设备相关位图依赖于显示设备的位模式，设备无关位图描述了图形的真实外形，而不是由某种特定显示设备所显示图形的内部表示。

有关位图及位图文件的结构很多，在此我们介绍 4 个重要的结构，即 BITMAPFILE-HEADER、BITMAPINFO、BITMAPINFOHEADER 和 RGBQUAD。

BITMAPFILEHEADER 结构的定义如下：

```
typedef    struct    tag BITMAPFILEHEADER{
    WORD             bfType;
    DWORD            bfSize;
    WORD             bfReserved1;
    WORD             bfReserved2;
    DWORD            bfOffBits;
```

```
} BITMAPFILEHEADER;
```

其中，bfType 说明文件为一个位图文件，表示位图文件的 ASCII 码为 BM，其 16 进制表示为 0x4D42。bfSize、bfReserved1 和 bfReserved2 一般不采用。bfOffBits 表示位图文件中至位图位数据的字节总数，即字节偏移量。

BITMAPINFOHEADER 结构的定义如下：

```
typedef    struct    tag BITMAPINFOHEADER {
    DWORD        biSize;
    DWORD        biWidth;
    DWORD        biHeight;
    WORD         biPlanes;
    WORD         biBitCount;
    DWORD        biCompression;
    DWORD        biSizeImage;
    DWORD        biXPelsPerMeter;
    DWORD        biYPelsPerMeter;
    DWORD        biClrUsed;
    DWORD        biClrImportant;
} BITMAPINFOHEADER;
```

其中，biSize 表示该结构的大小，为 sizeof(BITMAPINFOHEADER)。biWidth 表示以像素为单位的位图宽度。biHeight 表示以像素为单位的位图高度。biPlanes 表示目标设备的位平面数，置为 1。biBitCount 表示位图的每个像素的位数，可能为 1,4,8,24。biCompression 表示位图的被压缩类型。若位图未被压缩，则该值为 0；若为 1，则表示采用 RLE(Run Length Encoded) 8 压缩；若为 2，则表示采用 RLE 4 压缩。biSizeImage 表示被压缩位图的字节数。biXPelsPerMeter 表示以目标设备每米的像素个数来指定位图的水平分辨率。biYPelsPerMeter 表示以目标设备每米的像素个数来指定位图的垂直分辨率。biClrUsed 指定位图实际使用的颜色数。biClrImportant 指定对于显示该位图比较重要的颜色索引的个数。若该值为 0，则所有的颜色都是重要的。

RGBQUAD 结构的定义如下：

```
typedef    struct    tagRGBQUAD{
    BYTE         rgbBLUE;
    BYTE         rgbGREEN;
    BYTE         rgbRED;
    BYTE         rgbReserved;
} RGBQUAD;
```

其中，rgbBLUE 表示蓝色分量；rgbGREEN 表示绿色分量；rgbRED 表示红色分量。rgbReserved 的值为 0。

BITMAPINFO 结构把 BITMAPINFOHEADER 结构和一张颜色表结合在一起，其定义如下：

```
typedef    struct    tagBITMAPINFO{
```

```
    BITMAPINFOHEADER        bmiHeader;
    RGBQUAD                 bmiColors[1];
}BITMAPINFO;
```

关于位图,重要的结构还有(请查阅 Windows 在线帮助):

BITMAP	BITMAPCOREHEADER
BITMAPCOREINFO	COLORADJUSTMENT
DIBSECTION	RGBTRIPLE

对于 24 位位图,则没有颜色的索引表项,位图头信息之后紧跟位数据。为了简单起见,我们在此只解决 24 位位图的问题。对于 8 位位图,将获得的位图数据的索引值作一个向颜色表项的映射,就可得到解决。

10.3.2 创建位图类

下面我们将利用 Visual C++ 生成一个专门处理位图的类。该类的主要任务是:打开一个给定的位图文件;获得位图的重要信息,比如位图的宽度和高度等;将位图数据装入内存。其步骤如下:

(1) 为工程 Texture 添加一个类,其名为 CDib。

(2) 编辑文件 CDib.h

增加以下保护成员变量:

```
LPBITMAPFILEHEADER          m_pBmFileHeader;
LPBITMAPINFO                m_pBmInfo;
LPBITMAPINFOHEADER          m_pBmInfoHeader;
RGBQUAD *                   m_pRGBTable;
BYTE *                      m_pDibBits;
UINT                        m_numColors;
```

增加以下公有成员函数:

```
BYTE *                      GetDibBitsPtr();
LPBITMAPINFO                GetDIbInfoPtr();
LPBITMAPINFOHEADER          GetDibInfoHeaderPtr();
UINT                        GetDibHeight();
UINT                        GetDibWidth();
DWORD                       GetDibSizeImage();
CDib(const char * fileName);
```

添加保护成员函数如下:

```
void                        LoadBitmapFile(const char * fileName);
```

最后的文件 CDib.h 如下:

```
////////////////////////////        Dib.h        ////////////////////////////
////////////////////////////                     ////////////////////////////
//////////////////////////////////////////////////////////////////////////////////////////////

// Dib.h: interface for the CDib class.
//

#if ! defined(AFX_DIB_H__6C294E63_FE8B_11D0_9B6B_444553540000__INCLUDED_)
#define AFX_DIB_H__6C294E63_FE8B_11D0_9B6B_444553540000__INCLUDED_

#if _MSC_VER >= 1000
#pragma once
#endif // _MSC_VER >= 1000

class CDib : public CObject
{
public:
        BYTE*                   GetDibBitsPtr();
        LPBITMAPINFO            GetDIbInfoPtr();
        LPBITMAPINFOHEADER      GetDibInfoHeaderPtr();
        UINT                    GetDibHeight();
        UINT                    GetDibWidth();
        DWORD                   GetDibSizeImage();
        CDib();
        CDib(const char* fileName);
        virtual ~CDib();
protected:

        LPBITMAPFILEHEADER      m_pBmFileHeader;
        LPBITMAPINFO            m_pBmInfo;
        LPBITMAPINFOHEADER      m_pBmInfoHeader;
        RGBQUAD*                m_pRGBTable;
        BYTE*                   m_pDibBits;
        UINT                    m_numColors;

protected:
        void LoadBitmapFile(const char* fileName);

};

#endif // ! defined(AFX_DIB_H__6C294E63_FE8B_11D0_9B6B_444553540000__INCLUDED_)
```

(3) 编辑 CDib.cpp 文件

首先在其头部增加文件包含：

```
#include "stdafx.h"
#include "Dib.h"
#include "windowsX.h"
```

最后的 CDib.cpp 文件如下(如有费解之处,请查阅 Windows 在线帮助或有关位图编程的相关书籍):

```
//////////////////////////////////////////////////////////////////////////////////////////
//////////////////////////                                  //////////////////////////
//////////////////////////            Dib.cpp               //////////////////////////
//////////////////////////                                  //////////////////////////
//////////////////////////////////////////////////////////////////////////////////////////

#include "stdafx.h"
#include "Dib.h"
#include "windowsX.h"

#ifdef _DEBUG
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#define new DEBUG_NEW
#endif

//////////////////////////////////////////////////////////////////////
// Construction/Destruction
//////////////////////////////////////////////////////////////////////

CDib::CDib()
{
    m_pBmFileHeader = NULL;
    m_pBmInfo = NULL;
    m_pBmInfoHeader = NULL;
    m_pRGBTable = 0;
    m_pDibBits = 0;
}

CDib::CDib(const char * fileName)
{
    //装载位图
    LoadBitmapFile(fileName);
}


CDib::~CDib()
{
```

```
    GlobalFreePtr(m_pBmInfo);
}

void CDib::LoadBitmapFile(const char* fileName)
{
    //打开位图文件
    CFile file(fileName,CFile::modeRead);

    //将位图文件头读入内存
    BITMAPFILEHEADER bmFileHeader;
    file.Read((void*)&bmFileHeader,sizeof(bmFileHeader));

    //检查是否为位图
    if(bmFileHeader.bfType!=0x4d42)
    {
        AfxMessageBox("Not a bitmap file");
        m_pBmFileHeader=0;
        m_pBmInfo=0;
        m_pBmInfoHeader=0;
        m_pRGBTable=0;
        m_pDibBits=0;
        m_numColors=0;
    }
    else
    {
        DWORD fileLength=file.GetLength();
        DWORD dibSize=fileLength-sizeof(bmFileHeader);

        //分配全局内存
        BYTE* pDib=(BYTE*)GlobalAllocPtr(GMEM_MOVEABLE,dibSize);

        pDib=new BYTE[dibSize];

        file.Read(pDib,dibSize);
        file.Close();

        //初始化指向结构 BITMAPINFO 和 BITMAPINFOHEADER 的指针
        m_pBmInfo=(LPBITMAPINFO)pDib;
        m_pBmInfoHeader=(LPBITMAPINFOHEADER)pDib;

        //获得位图大小
        m_pBmInfoHeader->biSizeImage=GetDibSizeImage();
        m_pDibBits=pDib+m_pBmInfoHeader->biSize;
    }
```

```
}

DWORD CDib::GetDibSizeImage()
{
    if(m_pBmInfoHeader->biSizeImage==0)
    {
        //单行字节数,即宽度
        DWORD byteWidth=(DWORD)GetDibWidth();

        //高度
        DWORD height=(DWORD)GetDibHeight();

        DWORD imageSize=byteWidth*height;

        return imageSize;
    }
    else
        return m_pBmInfoHeader->biSizeImage;
}

UINT CDib::GetDibWidth()
{
    return      (UINT)m_pBmInfoHeader->biWidth;
}

UINT CDib::GetDibHeight()
{
    return      (UINT)m_pBmInfoHeader->biHeight;
}

LPBITMAPINFOHEADER CDib::GetDibInfoHeaderPtr()
{
    return      m_pBmInfoHeader;
}

LPBITMAPINFO CDib::GetDIbInfoPtr()
{
    return      m_pBmInfo;
}

BYTE* CDib::GetDibBitsPtr()
{
    return      m_pDibBits;
}
```

要注意,位图类 CDib 还只能将 24 位未压缩的位图装入内存,还不能解决 1,4,8 位等其他类型的位图,当然,将类 CDib 作一些改进以支持这些位图也是不难的(需作一个颜色表项映射和一个位图调色盘处理)。

10.3.3 将位图贴到物体上

现在,我们来完成最后的工作,将位图贴到物体的表面。由于类 CDib 已将位图数据读入内存,因而这项工作是较轻松的。

(1) 添加文件包含及公有成员变量

编辑 TextureView.h 头文件,增加文件包含:

```
#include "dib.h"
```

然后再增加一个公有成员变量以便获得位图的各种信息:

```
CDib*      m_pDib;
```

(2) 增加一个保护成员函数

现在,请添加一个保护成员函数 BitmapTexture(),并编辑如下:

```
void CTextureView::BitmapTexture()
{
    m_pDib = new CDib("valley.BMP");

    //    m_pDib= new CDib("wheat.BMP");
    //    m_pDib= new CDib("hust.BMP");
    //    m_pDib= new CDib("studio.BMP");
    //    m_pDib= new CDib("sunset.BMP");
    //    m_pDib= new CDib("brick.BMP");

    GLubyte* pTextureBits=(GLubyte*)m_pDib->GetDibBitsPtr();

    GLint width=m_pDib->GetDibWidth();
    GLint height=m_pDib->GetDibHeight();

    glPixelStorei(GL_UNPACK_ALIGNMENT,1);

    //    定义纹理:
    glTexImage2D(GL_TEXTURE_2D,0,3,width,height,
                 0,GL_BGR_EXT,GL_UNSIGNED_BYTE,pTextureBits);

    //    控制纹理:
```

```
    glTexParameterf(GL_TEXTURE_2D,GL_TEXTURE_WRAP_S,GL_CLAMP);
    glTexParameterf(GL_TEXTURE_2D,GL_TEXTURE_WRAP_T,GL_CLAMP);
    glTexParameterf(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,
                    GL_NEAREST);
    glTexParameterf(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,
                    GL_NEAREST);

    //    说明纹理贴图方式:
    glTexEnvf(GL_TEXTURE_ENV,GL_TEXTURE_ENV_MODE,GL_DECAL);

    //    启动纹理贴图:
    glEnable(GL_TEXTURE_2D);
    glShadeModel(GL_SMOOTH);

    //    定义纹理坐标和物体几何坐标:
    glBegin(GL_QUADS);
        glTexCoord2f(0.0f,0.0f);glVertex3f(-1.0f,-1.0f,0.0f);
        glTexCoord2f(0.0f,1.0f);glVertex3f(-1.0f,1.0f,0.0f);
        glTexCoord2f(1.0f,1.0f);glVertex3f(1.0f,1.0f,0.0f);
        glTexCoord2f(1.0f,0.0f);glVertex3f(1.0f,-1.0f,0.0f);
    glEnd();

    glDisable(GL_TEXTURE_2D);
}
```

在这里,我们首先要制作位图。制作一个位图最简单的方法是用 Windows 95/NT 自带的画笔去任意地绘制一些图形,最后在存储时确保文件类型为"24 位位图",可以给定任意一个名字,最后将其存储到存放工程 Texture 的目录"… \ Texture \ "之下。当重新生成一幅位图时,必须对语句:

```
m_pDib= new CDib("valley.bmp");
```

作出改动以便程序能将你的位图贴到物体上。为了产生较好的效果,一般要求位图的宽度和高度均能被 4 整除。

BitmapTexture()函数首先通过类 CDib 打开一个名为 Valley.bmp(256 x 256)的位图,并将其位图数据读入内存,获得指向位图数据的指针,最后将位图数据作为纹理数据进行纹理贴图,并最终映射到一个四边形上。

在程序中,我们实际用到了 Valley.bmp,但还备有 Wheat.bmp,Studio.bmp,Sunset.bmp,Brick.bmp ,Hust.bmp 等位图。除 Hust.bmp 外,其他位图均来自 3D Studio MAX R1.2 中所带的 JPG 位图,可通过 3D Studio MAX 先将其转换为 24 位位图。要注意,经 3D Studio MAX 转换的位图可能是被压缩的位图,因此可以再次通过 Windows 画笔进行转换。Hust.bmp 是直接在画笔中完成的。

在该函数中,一个最重要的语句是:

```
glTexImage2D(GL_TEXTURE_2D,0,3,width,height,
```

0,GL_BGR_EXT,GL_UNSIGNED_BYTE,pTextureBits);

由于 Windows 位图对颜色信息的存储顺序是 RGB 顺序的逆序,因此,应确保第七个参数为 GL_BGR_EXT,如果该参数为通常的 GL_RGB,则输出的位图的颜色将不会是正确的。

(3) 编辑 DrawScene()函数

在结构 switch(){... ....}中添加语句:

```
case DibTexture: BitmapTexture(); break;
```

(4) 运行程序

再次编译并运行程序,执行菜单 Type 中的项 Bitmap Texture。结果将一幅山谷画贴到了物体的表面,如图 10-2 所示:

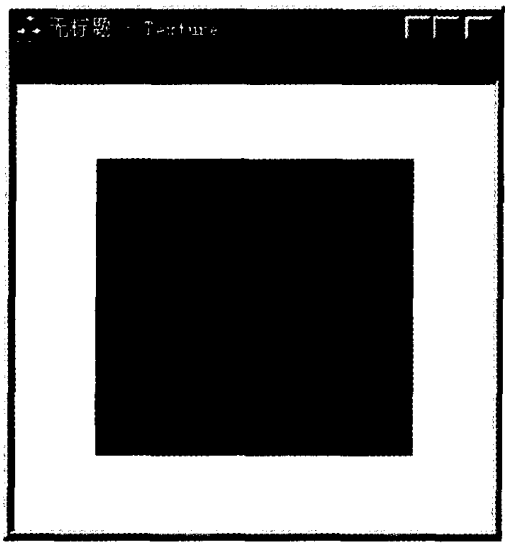

图 10-2 将位图贴到物体上

(5) 添加一个逻辑调色盘

按第 7 章中添加逻辑调色盘的方法给应用程序 Texture 添加一个逻辑调色盘。注意在拷贝时,对应位置的相应变动。

需要改动的地方位于 CMainFrame 类。首先在 MainFrm.cpp 文件中进行文件包含:

```
# include "TextureDoc.h"
# include "TextureView.h"
```

其次,确保 OnQueryNewPalette() 函数为:

```
BOOL CMainFrame::OnQueryNewPalette()
{
    // TODO: Add your message handler code here and/or call default

    WORD        i;
    CPalette    * pOldPal;

    CTextureView        * pView = (CTextureView * )GetActiveView();
```

```
    CClientDC     dc(pView);

    pOldPal = dc.SelectPalette(&pView->m_cPalette, FALSE);
    i = dc.RealizePalette();
    dc.SelectPalette(pOldPal, FALSE);

    if(i > 0)
        InvalidateRect(NULL);

    return CFrameWnd::OnQueryNewPalette();
}
```

(6) 运行程序

再次运行程序。尽管看起来并没有变化,但应用程序 Texture 会被更多机器所接受。

10.4 纹理坐标的自动生成

为了获得特殊效果(比如,将纹理贴到曲面上、观察物体轮廓、制作周围环境的反射效果等),有时需要自动生成纹理坐标而不是为物体的每个顶点均赋予纹理坐标。OpenGL 可以用 glTexGen()函数来产生自动纹理坐标,其原型(以 glTexGend 形式为例)为:

void glTexGend(GLenum coord, GLenum pname, GLdouble param);

其中,coord 参数可以为 GL_S、GL_T、GL_R 或 GL_Q,它们说明了纹理坐标 s,t,r,q 中哪一个坐标要自动生成。pname 参数的取值为 GL_TEXTURE_GEN_MODE。param 参数可以为 GL_OBJECT_LINEAR、GL_EYE_LINEAR 或 GL_SPHERE_MAP,它们决定用哪一个函数来生成纹理坐标。该函数共有 6 种不同的形式。

下面我们举一个例子来说明其应用。该例子是将一幅位图贴到一个茶壶上。茶壶是用辅助库函数生成的,这是本书唯一的一个地方用到 OpenGL 辅助库函数。通过该例可以学习如何将位图映射到一个曲面上。其制作步骤如下:

(1) 添加保护成员函数

添加一个保护成员函数 TexGenerate(),并编辑如下:

```
void CTextureView::TexGenerate()
{
    GLfloat currentCoeff[] = {1.0f,1.0f,1.0f,1.0f};

    m_pDib = new CDib("wheat.BMP");

    //    m_pDib = new CDib("brick.BMP");
    //    m_pDib = new CDib("valley.BMP");
    //.   m_pDib = new CDib("hust.BMP");
    //    m_pDib = new CDib("studio.BMP");
    //    m_pDib = new CDib("sunset.BMP");
```

```
GLubyte *     pTextureBits = (GLubyte * )m_pDib->GetDibBitsPtr();
GLint         width = m_pDib->GetDibWidth();
GLint         height = m_pDib->GetDibHeight();

glPixelStorei(GL_UNPACK_ALIGNMENT,1);

glTexEnvf(GL_TEXTURE_ENV,GL_TEXTURE_ENV_MODE,GL_MODULATE);
glTexParameterf(GL_TEXTURE_2D,GL_TEXTURE_WRAP_S,GL_REPEAT);
glTexParameterf(GL_TEXTURE_2D,GL_TEXTURE_WRAP_T,GL_REPEAT);
glTexParameterf(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,
                    GL_LINEAR);
glTexParameterf(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,
                    GL_LINEAR);
glTexImage2D(GL_TEXTURE_2D,0,3,width,height,
                    0,GL_BGR_EXT,GL_UNSIGNED_BYTE,pTextureBits);

//  纹理坐标的自动生成:
glTexGeni(GL_S,GL_TEXTURE_GEN_MODE,GL_SPHERE_MAP);
glTexGeni(GL_T,GL_TEXTURE_GEN_MODE,GL_SPHERE_MAP);
glTexGenfv(GL_S,GL_OBJECT_PLANE,currentCoeff);
glTexGenfv(GL_T,GL_OBJECT_PLANE,currentCoeff);

//  启动纹理贴图:
glEnable(GL_TEXTURE_GEN_S);
glEnable(GL_TEXTURE_GEN_T);
glEnable(GL_TEXTURE_2D);
glEnable(GL_CULL_FACE);
glEnable(GL_LIGHTING);
glEnable(GL_LIGHT0);
glEnable(GL_AUTO_NORMAL);
glEnable(GL_NORMALIZE);
glFrontFace(GL_CW);
glCullFace(GL_BACK);

glMaterialf(GL_FRONT,GL_SHININESS,32.0);
glPushMatrix();
    glRotatef(15.0f,1.0f,0.0f,0.0f);
    glRotatef(-5.0f,0.0f,1.0f,0.0f);
    auxSolidTeapot(0.8);
glPopMatrix();

glDisable(GL_TEXTURE_GEN_S);
glDisable(GL_TEXTURE_GEN_T);
```

```
    glDisable(GL_TEXTURE_2D);
    glDisable(GL_CULL_FACE);
    glDisable(GL_LIGHTING);
    glDisable(GL_LIGHT0);
    glDisable(GL_AUTO_NORMAL);
    glDisable(GL_NORMALIZE);
}
```

注意,在此应用了辅助库函数 auxSolidTeapot()。前面我们已经指出,应尽量少用该库中的函数。如果在你的产品应用程序中能够提供对辅助库函数的支持,当然就可以调用了。

(2) 编辑 DrawScene()函数

在结构 switch(){... ....}中添加语句:

```
case AutoTexture:        TexGenerate(); break;
```

(3) 包含头文件

在 TexturView.cpp 文件头部包含:

```
# include "GL \ glAux.h"
```

(4) 包含库文件

在 Visual C++开发平台上执行菜单:Project \ Settings... \ (或 Alt + F7),在弹出的对话框中选择 Link 标签,在 Object/libraray modules 中添加:glaux.lib。这样,就获得了对辅助库函数的支持。

(5) 运行程序

请再次编译连接并运行程序,执行菜单:Type \ AutoTexture,结果在画面中出现一个茶壶,而且带有纹理。当然纹理是反应你所采用的位图的。图 10-3 中的纹理位图是一块很大的麦浪,其背景为远处的树和蓝色的天空。

图 10-3 纹理坐标的自动生成

10.5 Mip 贴图

Mip 贴图是采用一系列不同大小的图像，当视线移近物体时，就以较高细节和较高清晰度的图像显示，反之就以较低细节和较模糊的图像显示，从而提高图像处理的整体效率。感谢 glTexImage2D()函数，我们可利用它完成 Mip 贴图。离视点越远，OpenGL 就用越小的图像来作贴图纹理。第零级是最大的，如果只有零级贴图，则 Mip 贴图就是通常的贴图，即第零级贴图所用的图像将贴满整个物体。如果贴图总数多于两级，则贴图就变为 Mip 贴图。一般地，第 n 级贴图所用的图像要比第 n-1 级贴图所用的图像要小，通常为其四分之一。为了充分体现 Mip 贴图的效果，还需将整个场景作放大处理。下面的程序完成 Mip 贴图处理。它将 6 个图贴到一个四边形上，这个四边形非常大。从最后的结果可以看出 Mip 贴图效果。按如下步骤完成：

(1) 添加成员变量

为 CTextureView 类添加两个保护成员变量：

```
GLint m_iWindowWidth;
GLint m_iWindowHeight;
```

并初始化如下：

```
m_iWindowWidth = 400;
m_iWindowHeight = 435;
```

注意，以上数据来源于 CMainFrame 类的成员函数 PreCreateWindow()中的语句：

```
cs.cx = 400;
cs.cy = 435;
```

如果 PreCreateWindow()函数中还没有这两条语句，请添加进去。如有请进行改动，这两个成员变量获取窗口的大小。

(2) 添加保护成员函数

为 CTextureView 类添加一个保护成员函数，并编辑如下：

```
void CTextureView::Mipmapping()
{
    int     i,j;
    GLubyte         mipmap32[32][32][4];
    GLubyte         mipmap16[16][16][4];
    GLubyte         mipmap8[8][8][4];
    GLubyte         mipmap4[4][4][4];
    GLubyte         mipmap2[2][2][4];
    GLubyte         mipmap1[1][1][4];

    //作 Mip 贴图的图像数据
    for (i = 0; i < 32; i++) {
```

```
for (j = 0; j < 32; j++) {
  mipmap32[i][j][0] = 50;
  mipmap32[i][j][1] = 125;
  mipmap32[i][j][2] = 255;
  mipmap32[i][j][3] = 255;
}
}
for (i = 0; i < 16; i++) {
for (j = 0; j < 16; j++) {
  mipmap16[i][j][0] = 200;
  mipmap16[i][j][1] = 200;
  mipmap16[i][j][2] = 0;
  mipmap16[i][j][3] = 255;
}
}
for (i = 0; i < 8; i++) {
for (j = 0; j < 8; j++) {
  mipmap8[i][j][0] = 255;
  mipmap8[i][j][1] = 0;
  mipmap8[i][j][2] = 0;
  mipmap8[i][j][3] = 255;
}
}
for (i = 0; i < 4; i++) {
for (j = 0; j < 4; j++) {
  mipmap4[i][j][0] = 0;
  mipmap4[i][j][1] = 200;
  mipmap4[i][j][2] = 0;
  mipmap4[i][j][3] = 255;
}
}
for (i = 0; i < 2; i++) {
for (j = 0; j < 2; j++) {
  mipmap2[i][j][0] = 0;
  mipmap2[i][j][1] = 0;
  mipmap2[i][j][2] = 200;
  mipmap2[i][j][3] = 255;
}
}
mipmap1[0][0][0] = 255;
mipmap1[0][0][1] = 255;
mipmap1[0][0][2] = 255;
mipmap1[0][0][3] = 255;
```

```
    glEnable(GL_DEPTH_TEST);
    glDepthFunc(GL_LESS);
    glClearColor(0.0f,0.0f,0.0f,1.0f);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glShadeModel(GL_FLAT);

    glPixelStorei(GL_UNPACK_ALIGNMENT, 1);

    //第零级
    glTexImage2D(GL_TEXTURE_2D, 0, 3, 32, 32, 0,
                 GL_RGBA, GL_UNSIGNED_BYTE, &mipmap32);

    //第一级
    glTexImage2D(GL_TEXTURE_2D, 1, 3, 16, 16, 0,
                 GL_RGBA, GL_UNSIGNED_BYTE, &mipmap16);

    //第二级
    glTexImage2D(GL_TEXTURE_2D, 2, 3, 8, 8, 0,
                 GL_RGBA, GL_UNSIGNED_BYTE, &mipmap8);

    //第三级
    glTexImage2D(GL_TEXTURE_2D, 3, 3, 4, 4, 0,
                 GL_RGBA, GL_UNSIGNED_BYTE, &mipmap4);

    //第四级
    glTexImage2D(GL_TEXTURE_2D, 4, 3, 2, 2, 0,
                 GL_RGBA, GL_UNSIGNED_BYTE, &mipmap2);

    //第五级
    glTexImage2D(GL_TEXTURE_2D, 5, 3, 1, 1, 0,
                 GL_RGBA, GL_UNSIGNED_BYTE, &mipmap1);

    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                 GL_NEAREST_MIPMAP_NEAREST);
    glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_DECAL);

    glEnable(GL_TEXTURE_2D);

    //放大视线,体现 Mip 贴图的作用
    gluPerspective(60.0, 1.0*(GLfloat)m_iWindowWidth/(GLfloat)m_iWindowHeight,
```

```
                1.0, 30000.0);

    //Mip贴图处理的物体
    glBegin(GL_QUADS);
        glTexCoord2f(0.0, 0.0); glVertex3f(-2.0, -1.0, 0.0);
        glTexCoord2f(0.0, 5.0); glVertex3f(-2.0, 1.0, 0.0);
        glTexCoord2f(5.0, 5.0); glVertex3f(1500.0, 1.0, -5000.0);
        glTexCoord2f(5.0, 0.0); glVertex3f(1500.0, -1.0, -5000.0);
    glEnd();

    glDisable(GL_TEXTURE_2D);

    //视线回位
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glFrustum(-1.0,1.0,-1.0,1.0,3.0,7.0);
}
```

最后指出,将如下语句

```
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                GL_NEAREST_MIPMAP_NEAREST);
```

更为

```
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                GL_LINEAR_MIPMAP_LINEAR);
```

将会获得平滑的 Mip 贴图处理。

(3) 编辑 OnSize()函数

为了时实地获取窗口大小变化的信息以便图像在原来的长宽比上呈比例地缩放,必须把长宽变比信息传递给变量 m_iWindowWidth 和变量 m_iWindowHeight。编辑 OnSize()函数,在语句 CView::OnSize(nType, cx, cy);后添加以下语句:

```
m_iWindowWidth = cx;
m_iWindowHeight = cy;
```

此时,OnSize()函数为:

```
void CTextureView::OnSize(UINT nType, int cx, int cy)
{
    CView::OnSize(nType, cx, cy);
    // TODO: Add your message handler code here
    m_iWindowWidth = cx;
    m_iWindowHeight = cy;

      if(cy > 0)
```

```
        if((m_oldRect.right > cx) || (m_oldRect.bottom > cy))
          RedrawWindow();
        m_oldRect.right = cx;
        m_oldRect.bottom = cy;
        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
          glFrustum(-1.0,1.0,-1.0,1.0,3.0,7.0);
          glViewport(0, 0, cx, cy);
      }
}
```

在此指出,在第 15 章我们还会介绍直接由 OpenGL 函数来获取窗口大小的方法。

(4) 编辑 DrawScene()函数

在结构 switch(){... ....}中添加语句:

```
case Mipmap:        Mipmapping();        break;
```

(5) 运行程序

再次编译连接并运行程序,执行菜单 Type \ Mip Mapping,最后见到整个 Mip 贴图处理的结果,如图 10-4。不难看出,最后一幅纹理图(1 × 1 位图)被放大处理了。整个场景是非常大的,但通过 Mip 贴图,简化了贴图处理的过程,提高了贴图效率。在进行 Mip 贴图时,由于采用近邻取样,因而各级 Mip 贴图图像间的过渡比较锐利,但执行速度较快。OpenGL 可以进行线性插值,以获得图像间的平滑过渡。目前,还不能直接完成双线性插值等高级技术。

图 10-4 Mip 贴图

可以建立纹理堆栈对纹理进行管理。纹理可以平移、旋转和缩放。在建立纹理堆栈之前,应调用 glMatrixMode()函数装载纹理矩阵。

还应该注意,物体的材质是物体的一种固有属性,给物体加上纹理并不等于改变了物体的

材质。比如,玻璃瓶总是由玻璃制成的,给玻璃瓶贴上标签只是为了说明其产品性能等信息。这个特点可以用来合成图像。比如,将 BitmapTexture()函数重新修改如下,就可获得一些特别的效果。

```
void CTextureView::BitmapTexture()
{
    /*//用于图像合成,需要采用融合处理技术,下一章将介绍这一重要方法。
    glEnable(GL_BLEND);
    glBlendFunc(GL_SRC_ALPHA,GL_ONE_MINUS_SRC_ALPHA);
    glColor4f(0.0f,1.0f,0.0f,0.5f);
    */
    m_pDib = new CDib("valley.BMP");

    GLubyte* pTextureBits = (GLubyte*)m_pDib->GetDibBitsPtr();

    GLint width = m_pDib->GetDibWidth();
    GLint height = m_pDib->GetDibHeight();

    glPixelStorei(GL_UNPACK_ALIGNMENT,1);
    glTexImage2D(GL_TEXTURE_2D,0,3,width,height,
                    0,GL_BGR_EXT,GL_UNSIGNED_BYTE,pTextureBits);

    glTexParameterf(GL_TEXTURE_2D,GL_TEXTURE_WRAP_S,GL_CLAMP);
    glTexParameterf(GL_TEXTURE_2D,GL_TEXTURE_WRAP_T,GL_CLAMP);
    glTexParameterf(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_NEAREST);
    glTexParameterf(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER, GL_NEAREST);
    glTexEnvf(GL_TEXTURE_ENV,GL_TEXTURE_ENV_MODE,GL_DECAL);

    glEnable(GL_TEXTURE_2D);
    glShadeModel(GL_SMOOTH);

    glBegin(GL_QUADS);
        glTexCoord2f(0.0f,0.0f);glVertex3f(-1.0f,-1.0f,0.0f);
        glTexCoord2f(0.0f,1.0f);glVertex3f(-1.0f,1.0f,0.0f);
        glTexCoord2f(1.0f,1.0f);glVertex3f(1.0f,1.0f,0.0f);
        glTexCoord2f(1.0f,0.0f);glVertex3f(1.0f,-1.0f,0.0f);
    glEnd();

    /*    //绘制第二幅用于合成的图像    */
    GLubyte polygonFillPattern[] = {
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x03, 0x80, 0x01, 0xC0, 0x06, 0xC0, 0x03, 0x60,
    0x04, 0x60, 0x06, 0x20, 0x04, 0x30, 0x0C, 0x20,
    0x04, 0x18, 0x18, 0x20, 0x04, 0x0C, 0x30, 0x20,
```

```
    0x04, 0x06, 0x60, 0x20, 0x44, 0x03, 0xC0, 0x22,
    0x44, 0x01, 0x80, 0x22, 0x44, 0x01, 0x80, 0x22,
    0x44, 0x01, 0x80, 0x22, 0x44, 0x01, 0x80, 0x22,
    0x44, 0x01, 0x80, 0x22, 0x44, 0x01, 0x80, 0x22,
    0x66, 0x01, 0x80, 0x66, 0x33, 0x01, 0x80, 0xCC,
    0x19, 0x81, 0x81, 0x98, 0x0C, 0xC1, 0x83, 0x30,
    0x07, 0xe1, 0x87, 0xe0, 0x03, 0x3f, 0xfc, 0xc0,
    0x03, 0x31, 0x8c, 0xc0, 0x03, 0x33, 0xcc, 0xc0,
    0x06, 0x64, 0x26, 0x60, 0x0c, 0xcc, 0x33, 0x30,
    0x18, 0xcc, 0x33, 0x18, 0x10, 0xc4, 0x23, 0x08,
    0x10, 0x63, 0xC6, 0x08, 0x10, 0x30, 0x0c, 0x08,
    0x10, 0x18, 0x18, 0x08, 0x10, 0x00, 0x00, 0x08};

    glPolygonMode(GL_FRONT_AND_BACK,GL_FILL);
    glPolygonStipple(polygonFillPattern);
    glEnable(GL_POLYGON_STIPPLE);

    glBegin(GL_POLYGON);
        glVertex3f(-1.0f,1.0f,0.005f);
        glVertex3f(-1.0f,-1.0f,0.005f);
        glVertex3f(1.0f,-1.0f,0.005f);
        glVertex3f(1.0f,1.0f,0.005);
    glEnd();
    glDisable(GL_POLYGON_STIPPLE);

    glDisable(GL_TEXTURE_2D);
}
```

如果需要合成两个图像,则需要应用融合处理方法,下一章我们将介绍这一重要方法。

第 11 章　增强图像效果

到目前为此，我们已知道怎样利用位图来增强图像效果，但是这远远不够用来模拟自然界中的各种纷繁现象。比如，怎样来模拟透明窗户玻璃以看清窗外自然界的美好景色；怎样生成光滑的流线型的轿车的外形；怎样模拟自然界的烟雾效果等等。对这些自然现象的模拟要求我们采用新的图形学技术。本章将介绍怎样解决这些问题的方法：

- 怎样融合物体，怎样生成透明物体；
- 怎样进行反走样处理；
- 怎样雾化场景。

11.1　OpenGL 融合

11.1.1　OpenGL 融合操作

融合(Blending)是一种最基本的图形图像处理技术，它是将两个或两个以上的物体或图像进行叠合最后生成新物体或一张新图像的过程。在图形图像处理中，为了产生特殊效果，通常要将一个物体挡住另一个物体或用一张图像盖在另一张图像的上面。当将一张图像放置在另一张图像的上面时，上面这张图像必须有某些部分是透明的，否则就无法看到下面一张图像的画面。

融合的方式有许多种，最主要的方式是使用 Alpha 通道。Alpha 通道是指加在 24 位真彩色图像上的另外 8 位信息，它用来描述 256 级不同的透明度数值。所以当将一张图像叠合在另一张图像上面时，在第二张图像中的 Alpha 信息便决定了如何将第二张图像的像素混合到第一张图像上，最后每个像素将产生一个新的颜色。两个物体的融合，则是通过目标物体与源物体颜色的组合产生新颜色的过程。对物体而言，Alpha 值决定物体颜色透明性。利用融合可生成透明效果。OpenGL 中，Alpha 值可归一化为区间[0.0,1.0]的任何一个值，Alpha 值越大则越不透明。利用 Alpha 信息不仅可以生成透明物体，而且可以避免锯齿状的发生，因为它会自动产生渐变色效果。当一个物体进行了消除锯齿状的操作以后，便会在其边缘加入过渡像素从而产生中间色效果，达到反走样的目的。3DS MAX 中正是充分地利用了这一图形图像处理技术才生成强大的融合透明处理工具。

在 OpenGL 中，要进行融合处理，首先要生成用于融合计算的两个因子，一个是源因子，另一个是目标因子。完成此功能的函数是 glBlendFunc()，其原型为：

```
void glBlendFunc( GLenum sfactor, GLenum dfactor );
```

其中，参数 sfactor 说明怎样计算源因子，参数 dfactor 说明怎样计算目标因子。sfactor 取值参见表 11-1。

表 11-1

常　量	最终源因子
GL_ZERO	(0,0,0,0)
GL_ONE	(1,1,1,1)
GL_DST_COLOR	(R_d, G_d, B_d, A_d)
GL_ONE_MINUS_DST_COLOR	$(1,1,1,1) - (R_d, G_d, B_d, A_d)$
GL_SRC_ALPHA	(A_s, A_s, A_s, A_s)
GL_ONE_MINUS_SRC_ALPHA	$(1,1,1,1) - (A_s, A_s, A_s, A_s)$
GL_DST_ALPHA	(A_d, A_d, A_d, A_d)
GL_ONE_MINUS_DST_ALPHA	$(1,1,1,1) - (A_d, A_d, A_d, A_d)$
GL_SRC_ALPHA_SATURATE	(f,f,f,1)

其中，$f = \min(A_s, 1 - A_d)$。dfactor 取值参见表 11-2。

表 11-2

常　量	最终目标因子
GL_ZERO	(0,0,0,0)
GL_ONE	(1,1,1,1)
GL_SRC_COLOR	(R_s, G_s, B_s, A_s)
GL_ONE_MINUS_SRC_COLOR	$(1,1,1,1) - (R_s, G_s, B_s, A_s)$
GL_SRC_ALPHA	(A_s, A_s, A_s, A_s)
GL_ONE_MINUS_SRC_ALPHA	$(1,1,1,1) - (A_s, A_s, A_s, A_s)$
GL_DST_ALPHA	(A_d, A_d, A_d, A_d)
GL_ONE_MINUS_DST_ALPHA	$(1,1,1,1) - (A_d, A_d, A_d, A_d)$

为了决定融合的最后结果，OpenGL 必须用源因子乘以源物体的颜色值，然后再用目标因子乘以目标物体的颜色值，最后再进行融合处理。

比如：

```
glBlendFunc(GL_ONE,GL_ZERO);
```

但是上述语句不能使 OpenGL 产生融合效果。因为源物体的每个点(事实上是像素)的颜色值乘以 1 再加上目标物体的同一点的颜色值乘以 0，所得颜色值也就是源物体的颜色值。

又如：

```
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_COLOR);
```

假设源物体的一个点的颜色值为(0.7,0.6,0.8,0.6)，目标物体在同一个像素点的颜色为(0.4,0.3,0.6,1.0)。OpenGL 首先获得源物体颜色向量中的 Alpha 值，为 0.6，用它作为融合的源因子(即每个分量的值均为 0.6)，然后乘以源物体的颜色向量中的各分量，得到(0.42,0.36,0.48,0.36)。紧接着 OpenGL 按 GL_ONE_MINUS_SRC_COLOR 计算用于融合的目标因子，为 0.4(即每个分量均为 0.4)，然后乘以目标物体的颜色向量中的各分量，得到

(0.16,0.12,0.24,0.4)。最后将两个颜色向量相加获得最终的颜色为(0.58,0.48,0.72,0.76)。

可见,OpenGL 融合操作可以总结为:设融合的源因子为(R_s,G_s,B_s,A_s),融合的目标因子为(R_d,G_d,B_d,A_d),源物体的颜色为(S_R,S_G,S_B,S_A),目标物体在同一像素点的颜色为(D_R,D_G,D_B,D_A),则最后融合的颜色为:

$$(R_s \times S_r + R_d \times D_R, G_s \times S_G + G_d \times D_G, B_s \times S_B + B_d \times D_B, A_s \times S_A + A_d \times D_A)$$

注意,颜色的各分量最后自动进行归一化处理,即大于 1.0 的值均被 1.0 所取代。

融合作为一种特殊的功能,必须进行启动和关闭申明:

```
glEnable(GL_BLEND);
glDisable(GL_BLEND);
```

11.1.2 融合四边形

下面,我们来对几个四边形进行融合处理,从中学习 OpenGL 融合处理的基本技术。可按以下步骤完成:

(1) 建立一个名为 Effect 的工程,并按第 5 章中的方法极小化 OpenGL。

(2) 添加枚举变量及保护成员变量

在 EffectView.h 文件的头部添加:

```
enum{Alpha2D, Alpha3D, AntialiasLine , AntialiasPlgn, AntialiasCube,Fog};
```

增加一个保护成员变量:

```
GLint m_typeControl;
```

初始化为:

```
m_typeControl = Alpha2D;
```

(3) 添加保护成员函数

添加一个保护成员函数:BlendQuads(),编辑如下:

```
void CEffectView::BlendQuads()
{
    glEnable(GL_BLEND);
    glBlendFunc(GL_SRC_ALPHA,GL_ONE_MINUS_SRC_ALPHA);
    glColor4f(0.0f,1.0f,0.0f,0.75f);
    glRectf(-0.7f,0.3f,0.3f,-0.7f);
    glColor4f(0.0f,0.0f,1.0f,0.5f);
    glPushMatrix();
        glTranslatef(0.0f,0.0f,0.05f);
        glRectf(-0.1f,1.0f,1.0f,-0.1f);
    glPopMatrix();
    glDisable(GL_BLEND) ;
```

该函数对两个四边形进行融合处理。第一个四边形的颜色为绿色,第二个四边形的颜色为蓝色,最后在两个四边形重叠的区域生成一个过渡色四边形。在 BlendQuads()函数中,glRectf()函数生成一个四边形,语句 glRectf(-0.7f,0.3f,0.3f,-0.7f);相当于:

```
glBegin(GL_POLYGON);
    glVertex2f(-0.7f,0.3f);
    glVertex2f(-0.7f,-0.7f);
    glVertex2f(0.3f,-0.7f);
    glVertex2f(0.3f,0.3f);
glEnd();
```

(4) 添加菜单

添加一个名为 Type 的菜单,即在属性对话框的 Caption 栏内加入 &Type。增加以下菜单项,表 11-3 列出了它们的属性对话框中的 Caption 及 ID 号。

表 11-3

Caption	ID
&Blending 2D	IDM_ALPHA_2D
B&lending 3D	IDM_ALPHA_3D
&Antialiasing Line	IDM_ANTI_LINE
A&ntialiasing Polygon	IDM_ANTI_PLGN
An&tialiasing Cube	IDM_ANTI_CUBE
&Fog Scene	IDM_FOG

按本书通用的方法响应上述诸消息。

```
void CEffectView::OnAlpha2d()
{
    // TODO: Add your command handler code here
    m_typeControl = Alpha2D;
    Invalidate();
}

void CEffectView::OnUpdateAlpha2d(CCmdUI* pCmdUI)
{
    // TODO: Add your command update UI handler code here
    if(m_typeControl == Alpha2D)
        pCmdUI->SetCheck(TRUE);
    else
        pCmdUI->SetCheck(FALSE);
}
```

```
void CEffectView::OnAlpha3d()
{
    // TODO: Add your command handler code here
    m_typeControl = Alpha3D;
    Invalidate();
}

void CEffectView::OnUpdateAlpha3d(CCmdUI* pCmdUI)
{
    // TODO: Add your command update UI handler code here
    if(m_typeControl == Alpha3D)
        pCmdUI->SetCheck(TRUE);
    else
        pCmdUI->SetCheck(FALSE);
}

void CEffectView::OnAntiLine()
{
    // TODO: Add your command handler code here
    m_typeControl = AntialiasLine;
    Invalidate();
}

void CEffectView::OnUpdateAntiLine(CCmdUI* pCmdUI)
{
    // TODO: Add your command update UI handler code here
    if(m_typeControl == AntialiasLine)
        pCmdUI->SetCheck(TRUE);
    else
        pCmdUI->SetCheck(FALSE);
}

void CEffectView::OnAntiPlgn()
{
    // TODO: Add your command handler code here
    m_typeControl = AntialiasPlgn;
    Invalidate();
}

void CEffectView::OnUpdateAntiPlgn(CCmdUI* pCmdUI)
{
    // TODO: Add your command update UI handler code here
    if(m_typeControl == AntialiasPlgn)
        pCmdUI->SetCheck(TRUE);
```

```
    else
        pCmdUI->SetCheck(FALSE);
}

void CEffectView::OnAntiCube()
{
    // TODO: Add your command handler code here
    m_typeControl = AntialiasCube;
    Invalidate();
}

void CEffectView::OnUpdateAntiCube(CCmdUI* pCmdUI)
{
    // TODO: Add your command update UI handler code here
    if(m_typeControl == AntialiasCube)
        pCmdUI->SetCheck(TRUE);
    else
        pCmdUI->SetCheck(FALSE);
}

void CEffectView::OnFog()
{
    // TODO: Add your command handler code here
    m_typeControl = Fog;
    Invalidate();
}

void CEffectView::OnUpdateFog(CCmdUI* pCmdUI)
{
    if(m_typeControl == Fog)
        pCmdUI->SetCheck(TRUE);
    else
        pCmdUI->SetCheck(FALSE);
}
```

(5) 编辑函数 DrawScene()

```
void CEffectView::DrawScene()
{
    glClearColor(1.0f,1.0f,1.0f,1.0f);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glTranslatef(0.0f,0.0f,-4.5f);
    switch(m_typeControl)
```

```
    }
        case Alpha2D:           BlendQuads(); break;
    }
    glFlush();
}
```

(6) 运行程序

编译连接并运行程序,最后在画面上出现两个经融合处理的四边形,如图 11-1。要注意,融合处理的顺序是非常重要的。请读者对调如下两语句的位置,看看运行结果有什么变化。为了看得较清楚,还可以改变两颜色向量的各分量:

```
glColor4f(0.0f,0.0f,1.0f,0.75f);
glColor4f(0.0f,1.0f,0.0f,0.75f);
```

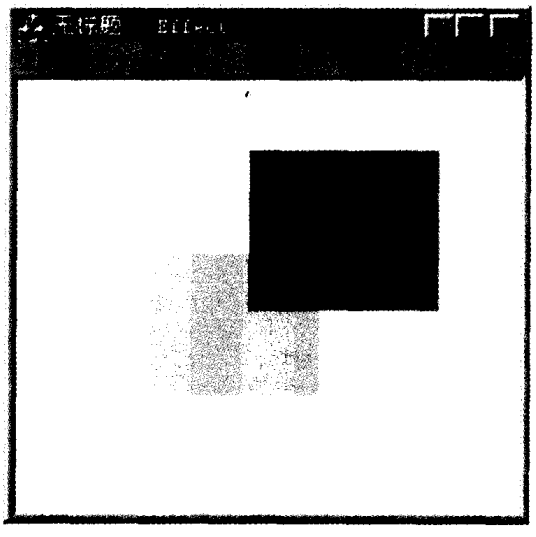

图 11-1 融合 2D 物体

11.1.3 制作透明玻璃

本节通过制作一块透明玻璃来说明如何制作透明物体。主要任务是通过透明玻璃看到后面的球体。

要制作三维的透明物体,简单地启动融合功能和给定融合因子是不够的,因为三维物体间有其位置的相互关系,体现这种关系的是物体离视点的远近。比如,玻璃离视点近,球离视点远。因而必须对深度进行检测和处理。在第 7 章,我们已经指出,OpenGL 缺省的深度测试条件是:离视点距离小于已画点至视点的距离则通过测试。因而透明物体和不透明物体的绘制顺序是至关重要的。一般来说应先画不透明的物体,因为透明物体的绘制需要应用 OpenGL 的融合功能,但是融合功能一旦启动,就会影响整个后继物体的着色。如果将不透明物体先画又会产生新的问题。因为透明体往往位于不透明体的前面,即离视点要近一些,所以透明体相应的点就会获得通过而覆盖已绘制的不透明体的对应部分。这时,OpenGL 采用了一个小技

巧,即一旦画完不透明的物体就立即将深度缓存进行屏蔽,即深度缓存变为只读模式,不能向深度缓存写进深度信息。这样就避免了已绘制的不透明物体的深度信息可能遇到的破坏。同时,OpenGL 还从深度缓存中读取深度信息,从而决定不透明物体的哪些单元需要进行融合处理。采用融合技术绘制三维物体的步骤是:

第一步:启动深度测试;

第二步:绘制不透明体;

第三步:设置深度缓存为只读模式,并启动融合;

第四步:绘制透明体;

第五步:恢复深度缓存的可读可写属性。

注意,在绘制透明体时,同样要进行深度比较,只是由于深度缓存是只读的,这样位于离视点更近的物体不会挡住已经被绘制的不透明体,而是和不透明体进行融合。

下面,我们按照这一步骤来完成本节的任务。

(1) 添加保护成员函数

添加如下 3 个保护成员函数:Sphere(),DrawCube()和 BlendObjects()。

Sphere()函数利用 OpenGL 实用库函数生成一个球,并用显示列表管理球,最后返回显示列表索引值。编辑如下:

```
GLint CEffectView::Sphere(GLdouble radius)
{
    //球
    GLint     sphereList = glGenLists(1);
    GLUquadricObj* quadObj;
    glNewList(sphereList, GL_COMPILE);
        quadObj = gluNewQuadric();
        gluQuadricDrawStyle(quadObj,GLU_FILL);
        gluQuadricNormals(quadObj,GLU_SMOOTH);
        gluSphere(quadObj,radius,32,32);
    glEndList();
    return sphereList;
}
```

(2) 编辑 DrawCube( )函数

DrawCube()函数负责不透明玻璃的生成,事实上它生成一个长方体,编辑如下:

```
void CEffectView::DrawCube(GLdouble dLength, GLdouble dWidth, GLdouble dHeight)
{
    glBegin(GL_QUAD_STRIP);
        glVertex3d(0.0, dHeight, dWidth);
        glVertex3d(0.0, 0.0, dWidth);
        glVertex3d(dLength, dHeight, dWidth);
        glVertex3d(dLength, 0.0, dWidth);
        glVertex3d(dLength, dHeight, 0.0);
        glVertex3d(dLength, 0.0, 0.0);
        glVertex3d(0.0, dHeight, 0.0);
```

```
        glVertex3d(0.0, 0.0,0.0);
        glVertex3d(0.0, dHeight, dWidth);
        glVertex3d(0.0, 0.0, dWidth);
    glEnd();
    glBegin(GL_QUADS);
        glVertex3d(0.0, dHeight, 0.0);
        glVertex3d(0.0, dHeight, dWidth);
        glVertex3d(dLength, dHeight, dWidth);
        glVertex3d(dLength, dHeight, 0.0);
    glEnd();
    glBegin(GL_QUADS);
        glVertex3d(0.0, 0.0, 0.0);
        glVertex3d(dLength, 0.0, 0.0);
        glVertex3d(dLength, 0.0, dWidth);
        glVertex3d(0.0, 0.0, dWidth);
    glEnd();
}
```

(3) 编辑 BlendObjects( )

BlendObjects()函数则负责球和玻璃的绘制,场景光照的生成,球和玻璃材质的赋予等等任务。BlendObjects()函数最后完成透明效果处理。编辑如下:

```
void CEffectView::BlendObjects()
{
    //光的位置
    GLfloat lightPosition[] = { 0.5f, 0.5f, 1.0f, 0.0f };
    //不透明物体材质
    GLfloat matSpecular[] = { 1.0f, 1.0f, 1.0f, 0.15f };
    GLfloat matShininess[] = { 100.0f };
    GLfloat matSolid[] = { 0.75f, 0.75f, 0.0f, 1.0f };
    GLfloat matZero[] = { 0.0f, 0.0f, 0.0f, 1.0f };
    //透明物体材质
    GLfloat matTransparent[] = { 0.0f, 0.8f, 0.8f, 0.6f };
    GLfloat matEmission[] = { 0.0f, 0.3f, 0.3f, 0.6f };
    glLightfv(GL_LIGHT0, GL_POSITION, lightPosition);
    //球
    GLint sphereList=Sphere(0.7);
    //玻璃
    GLint cubeList = glGenLists(1);
    glNewList(cubeList, GL_COMPILE);
        DrawCube(2.0f,0.01f,1.6f);
    glEndList();

    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
```

```
    glClearColor(0.0f,0.0f,0.0f,1.0f);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    //第一步:启动深度测试
    glEnable(GL_DEPTH_TEST);
    //第二步:绘制不透明体
    glPushMatrix ();
        glTranslatef (0.0f, 0.0f, 0.05f);
        glMaterialfv(GL_FRONT, GL_SPECULAR, matSpecular);
        glMaterialfv(GL_FRONT, GL_SHININESS, matShininess);
        glMaterialfv(GL_FRONT, GL_EMISSION, matZero);
        glMaterialfv(GL_FRONT, GL_DIFFUSE, matSolid);
        glCallList(sphereList);
    glPopMatrix();
    //第三步:设置深度缓存为只读模式,并启动融合
        glDepthMask(GL_FALSE);
        glEnable(GL_BLEND);
        glBlendFunc(GL_SRC_ALPHA, GL_ONE);
    //第四步:绘制透明体
    glPushMatrix();
        glTranslatef (-0.8f, -0.8f, 1.0f);
        glRotatef(5.0f, 1.0f, 1.0f, 0.0f);
        glRotatef(10.0f, 0.0f, 1.0f, 0.0f);
        glMaterialfv(GL_FRONT, GL_EMISSION, matEmission);
        glMaterialfv(GL_FRONT, GL_DIFFUSE, matTransparent);
        glCallList(cubeList);
    glPopMatrix();
    //第五步:恢复深度缓存的可读可写属性
        glDepthMask(GL_TRUE);
        glDisable(GL_BLEND);
        glDisable(GL_LIGHTING);
}
```

BlendObjects()函数没有用到新的函数,因而对其理解请按注释由自己完成。

(4) 编辑 DrawScene()函数

在结构 switch(){... ....}中添加语句:

```
case Alpha3D:       BlendObjects();       break;
```

注意,在编辑 DrawScene()函数之前,确保已对 IDM_ALPHA_3D 消息进行了响应。

(5) 运行程序

重新编译连接并运行程序,并执行菜单 Type \ Blending 3D,看看最后绘制的结果,如图 11-2 所示。由于要进行三维融合处理,计算花费较大。

为了制作更加高级的透明效果(比如,只将一幅图的人像重叠到背景图上去),需要进行 Alpha 测试。Alpha 测试通过比较同一像素的 Alpha 值来决定是否修改在帧缓存中已绘制的

像素。首先,为一幅图的不同部分设置不同的 Alpha 值;接着用 glAlpha()函数设置 Alpha 测试条件;最后利用 Alpha 信息、深度信息、场景和绘制条件进行处理。应该指出,进行 Alpha 测试和进行深度测试的过程及相关含义是一致的。

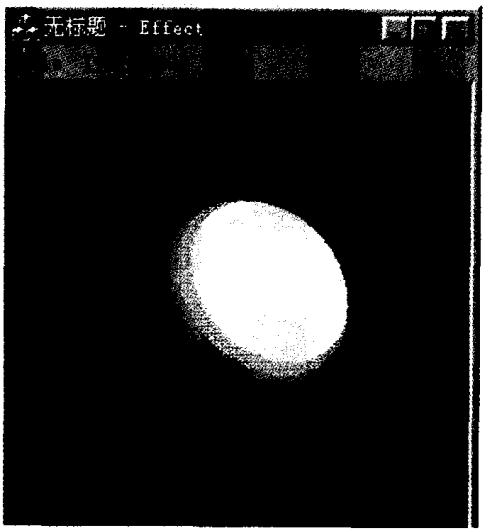

图 11-2 融合 3D 物体

11.2 反走样

11.2.1 反走样思想

在 Windows 画笔中绘制的直线往往很不光滑,因而不能满足用户对高精度图片的要求。在 Corel Draw 工具中,绘制的直线,无论其怎样倾斜,总是十分光滑的。直线不光滑,我们说它走样了。由于显示器的屏幕被离散为行列单元(即像素),也不可能再对每个小单元进行分割,因此,每个单元总是被一种颜色所填充。另一方面,一条直线首先要被离散化,然后填充尽可能与直线接近的像素来显示这条直线。很显然,离散得越细逼近程度就会越高,但是屏幕的像素资源是极其有限的,显示器的点距越小,硬件设计的难度就会越高,因而通过硬件直接显示的一条直线不可能是十分光滑的。为了解决这个问题,则产生了反走样技术。反走样技术事实上是一种图形学算法。图 11-3 是一条没有经过反走样的直线。

反走样就是要在锯齿直线的边缘填充一些像素,这些像素的颜色则是一种过渡颜色。过渡色的选取是由直线的颜色和直线周围像素的颜色来决定的,比如:如果直线的颜色为黑色而直线周围的颜色为白色,则可以选择一个灰色作为周围像素的填充颜色。这样从整体上看,直线的严重锯齿情况就会得到改善。这是一个非常好的思想。当然,实际的反走样过程要比这个思想复杂一些。图 11-4 展示了反走样的过程。

11.2.2 OpenGL 反走样

OpenGL 不仅能对直线进行反走样,而且也能对多边形进行反走样。OpenGL 提供了不同

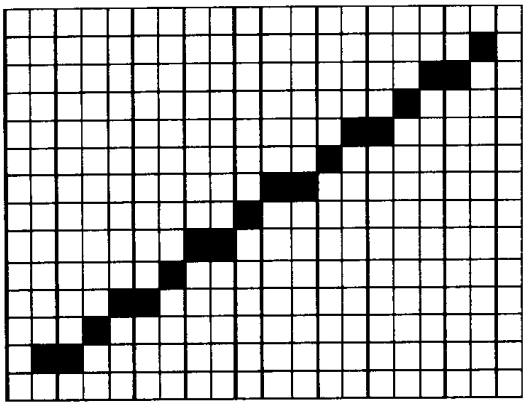

图 11-3　走样直线

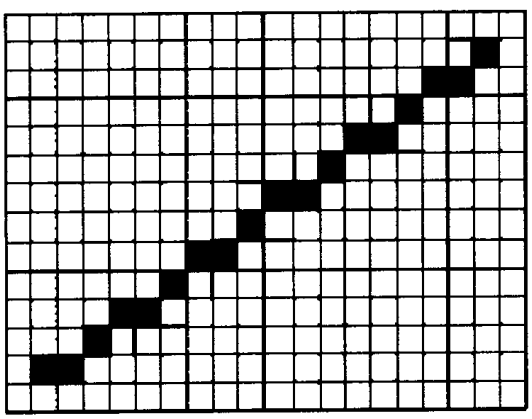

图 11-4　光顺直线

级别的反走样算法,采用何种级别的算法是由 OpenGL 的行为控制函数 glHint()来控制的。该函数的原型为:

void glHint(GLenum target, GLenum mode);

其中,target 参数控制操作行为,mode 参数说明执行控制行为的方式。target 参数的取值参见表 11-4。

表 11-4

常　量	含　义
GL_FOG_HINT	雾的计算精度
GL_LINE_SMOOTH_HINT	直线的反走样采样质量
GL_PERSPECTIVE_CORRECTION_HINT	颜色及纹理的插值质量
GL_POINT_SMOOTH_HINT	点的反走样采样质量
GL_POLYGON_SMOOTH_HINT	多边形的反走样采样质量

mode 参数的取值可见表 11-5。

表 11-5

常 量	含 义
GL_FASTEST	给出最有效选择
GL_NICEST	给出最好的选择
GL_DONT_CARE	不许客户选择

OpenGL 在进行反走样操作时，首先获取 glHint( )函数所提供的反走样行为，然后 OpenGL 按照当前实际的运行状况决定 OpenGL 内核的反走样算法。比如，GL_PERSPECTIVE_CORRECTION_HINT 它告诉 OpenGL 怎样对颜色和纹理坐标进行插值。有两种插值方式可供选择，即要么选择按像素值进行插值，要么选择按透视校正法进行插值。在执行的过程中，如果 OpenGL 感到力不从心，则 OpenGL 会简单地进行线性插值计算。

在进行反走样之前，要告诉 OpenGL 启动反走样功能。对点进行反走样时，要执行下述语句：

```
glEnable(GL_POINT_SMOOTH);
```

对直线而言，则为：

```
glEnable(GL_LINE_SMOOTH);
```

对多边形则为：

```
glEnable(GL_POLYGON_SMOOTH);
```

反走样还需要结合融合功能，这样，OpenGL 就会知道按什么样的公式去计算颜色值。颜色值则来自于被反走样的物体和物体周围的颜色。

11.2.3 反走样的线框球和三角形

本节介绍怎样对物体进行反走样。其任务是反走样一个线框球、一个三角形和一个盒子。可按以下步骤完成：

(1) 添加保护成员函数

添加一个保护成员函数：AntialiasLineSphere()，编辑如下：

```
void CEffectView::AntialiasLineSphere()
{
    glClearColor(0.0f,0.0f,0.0f,1.0f);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    //光
    GLfloat lightPosition[] = { 0.5f, 0.5f, 1.0f, 0.0f };
    glLightfv(GL_LIGHT0, GL_POSITION, lightPosition);

    //材质
    GLfloat matSpecular[] = { 1.0f, 1.0f, 1.0f, 0.0f };
```

```
    GLfloat matShininess[] = { 100.0f };
    GLfloat matSolid[] = { 1.0f, 1.0f, 1.0f, 1.0f };
    GLfloat matZero[] = { 0.0f, 0.0f, 0.0f, 1.0f };
    glMaterialfv(GL_FRONT, GL_SPECULAR, matSpecular);
    glMaterialfv(GL_FRONT, GL_SHININESS, matShininess);
    glMaterialfv(GL_FRONT, GL_EMISSION, matZero);
    glMaterialfv(GL_FRONT, GL_DIFFUSE, matSolid);

    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);

    //启动线反走样功能
    glEnable(GL_LINE_SMOOTH);

    //利用融合
    glEnable(GL_BLEND);
    glBlendFunc(GL_SRC_ALPHA,GL_ONE_MINUS_SRC_ALPHA);

    //行为控制
    glHint(GL_LINE_SMOOTH_HINT,GL_NICEST);
    glLineWidth(1.0f);

    //线型球
    GLint sphereList = glGenLists(1);
    GLUquadricObj * quadObj;
    glNewList(sphereList, GL_COMPILE);
        quadObj = gluNewQuadric();
        gluQuadricDrawStyle(quadObj,GLU_LINE);
        gluQuadricNormals(quadObj,GLU_SMOOTH);
        gluSphere(quadObj,0.9,16,16);
    glEndList();

    glCallList(sphereList);

    glDisable(GL_LINE_SMOOTH);
    glDisable(GL_BLEND);
    glDisable(GL_LIGHTING);
}
```

(2) 编辑 DrawScene()函数

在结构 switch(){... ....}中添加语句:

```
case AntialiasLine:        AntialiasLineSphere();   break;
```

(3) 运行程序

重新编译连接并运行程序，并执行菜单 Type \ Antialiasing Line，可见，线是较光滑的，如图 11-5。如果关闭反走样功能，则线的锯齿状加剧。

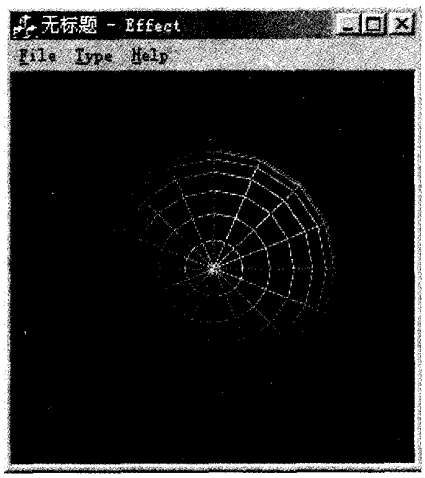

图 11-5　反走样线

现在，我们来对一个三角形进行反走样处理。对于单个三角形，反走样事实上就是光滑其边缘。

(1) 添加保护成员函数

添加一个保护成员函数：AntialiasTriangle()，编辑如下：

```
void CEffectView::AntialiasTriangle()
{
    //启动多边形反走样功能
    glEnable(GL_POLYGON_SMOOTH);

    //利用融合
    glEnable(GL_BLEND);
    glBlendFunc(GL_SRC_ALPHA,GL_ONE_MINUS_SRC_ALPHA);

    //行为控制
    glHint(GL_POLYGON_SMOOTH_HINT,GL_NICEST);

    glPushMatrix();
        glRotatef(35.0f, 0.0f, 0.0f, 1.0f);

        glBegin(GL_TRIANGLES);
            glColor4f(0.2f,0.5f,1.0f,0.8f);
            glVertex2d(-0.8, 0.8);
            glVertex2d(-0.8, -0.8);
            glVertex2d(0.8, -0.8);
```

```
        glEnd();
    glPopMatrix();

    glDisable(GL_POLYGON_SMOOTH);
    glDisable(GL_BLEND);
}
```

(2) 编辑 DrawScene()函数

在 switch(){... ....}中添加语句:

```
    case AntialiasPlgn:    AntialiasTriangle();    break;
```

(3) 运行程序

再次编译连接并运行程序,执行菜单 Type \ Antialiasing Polygon,看看最后绘制的结果,如图 11-6 所示。可见,三角形边缘是较光滑的。如果关闭反走样功能,则边缘线的锯齿状加剧。

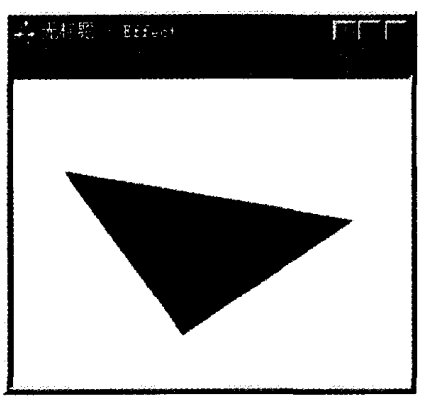

图 11-6 反走样多边形

在进行多边形反走样操作时,首先,面的绘制顺序是十分重要的。一般来说,应该先绘制与视点最近的面,最后绘制与视点最远的面。其次,用于融合的源因子、目标因子和 Alpha 值将极大地影响反走样的结果。并且,在反走样结束时,相重合的边有可能出现不正常的问题。比如,在 AntialiasTriangle()函数中,如果用绘制一个正方形的语句来代替该函数中绘制三角形的相关语句,运行程序,结果出现了一条本不需要的白色对角线。这些均与因子的选择有关。最后,应在精度与速度间作出调整,因为多边形反走样功能需要花费大量的计算时间。

下面,我们来反走样一个盒子。请按如下步骤操作:

(1) 添加保护成员函数

添加一个保护成员函数:AntiCube() ,编辑如下:

```
void CEffectView::AntiCube()
{
    //启动多边形反走样功能
    glEnable(GL_POLYGON_SMOOTH);
```

```
    //利用融合
    glEnable(GL_BLEND);
    glBlendFunc(GL_SRC_ALPHA,GL_ONE_MINUS_SRC_ALPHA);

    //行为控制
    glHint(GL_POLYGON_SMOOTH_HINT,GL_NICEST);

    glPushMatrix();
      glTranslatef(-0.3f,-0.5f,0.0f);
      glRotatef(35.0f, 0.0f, 0.0f, 1.0f);
      glRotatef(20.0f, 1.0f, 0.0f, 0.0f);
      glRotatef(20.0f, 0.0f, 1.0f, 0.0f);

      DrawCube(1.0f,1.0f,1.0f);
    glPopMatrix();

    glDisable(GL_POLYGON_SMOOTH);
    glDisable(GL_BLEND);
}
```

(2) 编辑 DrawScene()函数

在结构 switch(){... ....}中添加语句:

```
case AntialiasCube:   AntiCube();          break;
```

(3) 运行程序

再次编译连接并运行程序,执行菜单 Type \ Antialiasing Cube,看看最后绘制的结果,如图 11-7所示。可见,一个盒子得到了光顺处理,但从画面上看,又产生了另一个问题:盒子的边缘出现了一些点没有绘制的情况,而且盒子的面的对角线也出现了同一现象,同时有两个面的颜色与其他面的颜色有很大的不同。请从融合的顺序、背景颜色及深度测试条件出发考虑这个问题。另外,要解决上述问题,可以将构成盒子的边进行重新绘制,高级处理技术需要进行边缘测试处理。

11.3 OpenGL 雾化

许多视景都需要较强的真实感和空间感。例如,早春和晚秋的夜景可以用雾来生成逼真的氛围。环境可能为大地或海洋、白昼或黑夜,环境中的物体自视线由近及远逐渐模糊。这些模糊的景物由近及远最终淡化为雾状物体。雾化效果是 3D 图形图像处理的常用特性,在游戏中常见的烟雾、爆炸的火焰及白云等效果都是雾化的结果。雾化不仅可以保证远景的真实性,而且还可以减少 3D 图形渲染的工作量。

OpenGL 底层化了所有的与雾化处理有关的图形学算法,可以通过函数直接控制雾的属性,如颜色、雾的浓度、雾的某些行为属性。雾的行为属性包括雾的影响范围、雾的颗粒大小、雾的地平线杂波属性、雾的活跃程度、雾的卷须大小等等。应用 OpenGL 雾化处理函数还可以

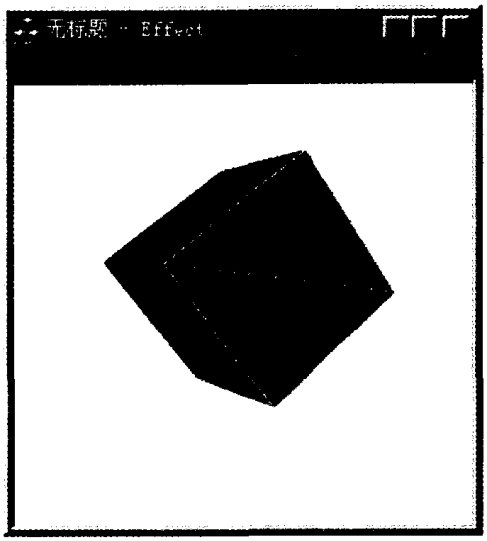

图 11-7　反走样盒子

模拟容积光。容积光是点光源透过雾或蒸气照射的光，这种光对于增加氛围和真实感非常有用。电影 Batman 中的点光源效果就是应用容积光来完成的。

本节，我们将学会雾化操作的基本步骤，并明确怎样应用 OpenGL 雾化来增强环境效果。

11.3.1　确定雾的属性

在 OpenGL 中雾的各种属性是通过 glFog()函数来设置的。该函数有 4 种不同的形式，它们是：

```
void glFogf( GLenum pname, GLfloat param );
void glFogi( GLenum pname, GLint param );
void glFogfv( GLenum pname, const GLfloat * params );
void glFogiv( GLenum pname, const GLint * params );
```

其中，pname 参数指明雾的属性。其取值可参见表 11-6。

表 11-6

常　量	含　义
GL_FOG_MODE	雾的计算模式
GL_FOG_DENSITY	雾的浓度
GL_FOG_START	线性雾方程中的近距离
GL_FOG_END	线性雾方程中的远距离
GL_FOG_COLOR	雾的颜色
GL_FOG_INDEX	雾的颜色，用于颜色索引模式

当 pname 参数为 GL_FOG_MODE 时，需要 params 参数提供用于雾计算的融合因子，只能选取 GL_LINEAR，GL_EXP，GL_EXP2 三者之一。当然，params 还可能为一个浮点数、一个

整数或一个向量。下面几条语句都用来描写雾的属性：

```
GLfloat fogColor[4] = {1.0, 1.0, 1.0, 1.0};
glFogi (GL_FOG_MODE, GL_LINEAR);
glFogfv (GL_FOG_COLOR, fogColor);
glFogf (GL_FOG_DENSITY, 0.35);
glFogf (GL_FOG_START, 1.0);
glFogf (GL_FOG_END, 7.0);
```

在 OpenGL 中，雾化使用融合因子 f 把雾化色同光栅化基片的后纹理(Posttexture)相融合。因子 f 的计算按以下公式进行：

若参数 params 为 GL_LINEAR，则 $f=\frac{end-z}{end-start}$；若为 GL_EXP，则 $f=e^{-(disant\times z)}$；若为 GL_EXP2，则 $f=e^{-(disant\cdot z)\times(disant\cdot z)}$。

在 RGBA 模式中，如果 C_r 代表光栅化基片的颜色向量，C_f 代表雾的颜色向量，则雾化产生的对应值 C 为：$C=fC_r+(1-f)C_f$。

雾化必须用行为控制函数 glHint()来提供计算线索，也必须激活和挂起雾化功能。

11.3.2 雾化场景

下面让我们来雾化一个场景，这个场景是由一些球和盒子组成的。

(1) 添加保护成员函数

添加一个保护成员函数 FogScene()，并编辑如下：

```
void CEffectView::FogScene()
{
    glClearColor(0.5, 0.5, 0.5, 1.0);

    //光照与材质
    GLfloat position[] = { 0.5, 0.5, 3.0, 0.0 };
    glLightfv(GL_LIGHT0, GL_POSITION, position);

    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    {
        GLfloat material[3] = {0.2f, 0.05f, 0.05f};

        glMaterialfv (GL_FRONT, GL_AMBIENT, material);

        material[0] = 0.8f;     material[1] = 0.1f;     material[2] = 0.05f;
        glMaterialfv (GL_FRONT, GL_DIFFUSE, material);

        material[0] = 0.8f;     material[1] = 0.8f;     material[2] = 0.8f;
        glMaterialfv (GL_FRONT, GL_SPECULAR, material);
```

```
    glMaterialf (GL_FRONT, GL_SHININESS, 60.0);
}

//雾化
glEnable(GL_FOG);
{
    GLfloat fogColor[4] = {1.0, 1.0, 1.0, 1.0};
    glFogi (GL_FOG_MODE, GL_LINEAR);
    glFogfv (GL_FOG_COLOR, fogColor);
    glFogf (GL_FOG_DENSITY, 0.35);
    glFogf (GL_FOG_START, 1.0);
    glFogf (GL_FOG_END, 7.0);
    glHint (GL_FOG_HINT, GL_DONT_CARE);
}

//场景
GLint sphere1 = Sphere(0.6);
GLint sphere2 = Sphere(0.4);
GLint sphere3 = Sphere(0.3);
GLint sphere4 = Sphere(0.2);

glPushMatrix();
    glTranslatef(0.0f, -0.5f,0.0f);
    glCallList(sphere1);

    glTranslatef(-1.0f,0.0f, -0.0f);
    glCallList(sphere3);

    glTranslatef(2.0f,0.0f,0.0f);
    glCallList(sphere3);

    glTranslatef(0.5f,0.5f, -2.0f);
    glCallList(sphere2);

    glTranslatef(-1.0f,0.0f,0.0f);
    glCallList(sphere4);

    glTranslatef(-1.6f,0.0f,0.0f);
    glRotatef(35.0f, 1.0f, 1.0f, 0.0f);
    glRotatef(10.0f, 0.0f, 1.0f, 0.0f);
    DrawCube(1.0f,1.0f,1.f);

    glTranslatef(1.6f,0.0f,0.0f);
    DrawCube(0.5f,1.0f,1.f);
```

```
    glPopMatrix();

    glDisable(GL_LIGHTING);
    glDisable(GL_FOG);
}
```

(2) 编辑 DrawScene()函数

在结构 switch(){... ....}中添加语句:

```
case Fog: FogScene(); break;
```

(3) 运行程序

重新编译连接并运行程序,执行菜单 Type \ Fog Scene,最后绘制的结果如图 11-8 所示。本来,场景中所有物体的材质均为红色,且场景中只有一盏白色的灯光。由于雾化效果,球和盒子逐渐被雾化为背景色,效果是十分明显的。

如果按如下方式定义雾的属性,其结果将会发生很大的变化:

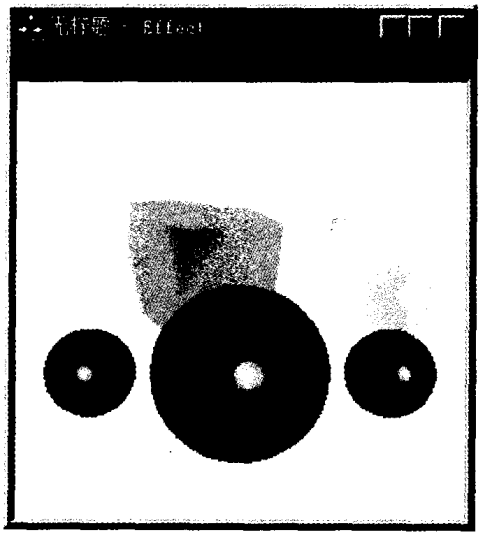

图 11-8 雾化场景

```
GLfloat fogColor[4] = {1.0, 1.0, 1.0, 1.0};
glHint (GL_FOG_HINT, GL_DONT_CARE);
glFogi (GL_FOG_MODE, GL_EXP);
glFogfv (GL_FOG_COLOR, fogColor);
glFogf (GL_FOG_DENSITY, 0.5);
glFogf (GL_FOG_START, 1.0);
glFogf (GL_FOG_END, 5.0);
```

或

```
GLfloat fogColor[4] = {1.0, 1.0, 1.0, 1.0};
```

```
glHint(GL_FOG_HINT, GL_DONT_CARE);
glFogi (GL_FOG_MODE, GL_EXP2);
glFogfv (GL_FOG_COLOR, fogColor);
glFogf (GL_FOG_DENSITY, 0.15);
glFogf (GL_FOG_START, 1.0);
glFogf (GL_FOG_END, 5.0);
```

第 12 章　实用库函数

设计 OpenGL 的一个指导原则就是提供程序的可移植性，而不是必须描述任何高级的图形对象。结果，基本的 OpenGL 接口不支持传统的同图形标准系统有关的一些几何对象。作为 OpenGL 基本图形系统的补充，在每个 OpenGL 系统实现时，都提供了 OpenGL Utillity Library。OpenGL Graphics System Utility Library 是一套专门补充 OpenGL 图形系统的子程序。它支持 Mip 贴图（Mip Mapping）、矩阵操作（Matrix Manipulation）、多边形细化（Polygon Tessellation）、二次曲面（Quadrics Surface）、非均匀有理样条（NURBS）以及错误处理。Mip 贴图包括图像变比以及自动 Mip 贴图生成。大量的矩阵操作函数建立投影矩阵和观察矩阵。多边形细化将凹多边形转化为三角形以便于描绘。二次曲面支持基本的二次曲面的绘制，如球体、圆锥体等。NURBS 曲线和曲面及被剪切的表面映射成比较简单的 OpenGL 评价器。本章将介绍：

- 如何生成标准的二次曲面；
- 如何进行图像处理；
- 如何进行自动 Mip 贴图。

12.1　标准二次曲面的生成

12.1.1　创建二次曲面对象

OpenGL 实用库函数可以按用户指定的多种方式绘制球体、圆筒和圆盘等。为了绘制二次曲面，首先应创建二次曲面对象，这个对象包含一个二次曲面应被绘制的方式和状态等信息；其次使用属性函数修改状态；最后绘制一个指定的二次曲面。现在，我们对第 7 章中绘制球的函数进行解释。

```
GLUquadricObj * quadObj;
quadObj = gluNewQuadric();
gluQuadricDrawStyle(quadObj, GLU_FILL);
gluQuadricNormals(quadObj, GLU_SMOOTH);
gluSphere(quadObj, 0.7, 32, 32);
```

在上述语句中，首先申明一个类型为 GLUquadricObj 的指针，数据类型 GLUquadricObj 表示一个二次曲面对象。gluNewQuadric()函数则生成一个二次曲面对象，并返回指向所生成的二次曲面的指针。

gluQuadricDrawStyle()函数则用于说明二次曲面的绘制方式。该函数有两个参数，第一个参数必须是一个类型为 GLUquadricObj 的指针，该指针指向一个二次曲面对象，而二次曲面是由 gluNewQuadric()函数所创建的。该函数的第二个参数为一个 GLU 常量，它可以为 GLU

_FILL、GLU _LINE、GLU _SILHOUETTE 和 GLU _POINT 中的任何一个。GLU _FILL 表示实体填充二次曲面；GLU _LINE 表示只绘制构成二次曲面的线框；GLU _SILHOUETTE 则绘制构成二次曲面的线框，但某些边界线可能不会得到渲染；GLU _POINT 表示仅仅绘制二次曲面的几何顶点。

gluQuadricNormals()函数生成二次曲面的法线信息。该函数有两个参数，第一个参数与 gluQuadricDrawStyle()函数的第一个参数是一样的。第二个参数也是一个 GLU 常量，它可以取如下三者之一：GLU _NONE、GLU _FLAT 和 GLU _SMOOTH。GLU _NONE 表示不生成法线。如果不生成法线，则光照计算有可能不正确。GLU _FLAT 则为二次曲面的每一个面都生成法线。GLU _SMOOTH 则为二次曲面的每一个顶点都生成法线。因为二次曲面每个顶点的法线都不会相同，这就给光照计算提供了最充分的信息。

与二次曲面状态有关的函数还有 gluQuadricOrientation()和 gluQuadricTexture()函数。gluQuadricOrientation()函数说明插值计算的方向。该函数共有两个参数。第一个参数是一个二次曲面对象的指针。第二个参数为一个 GLU 常量，它表示二次曲面插值计算的方向，该方向影响二次曲面的绘制。可以取如下二者之一：GLU _OUTSIDE 和 GLU _INSIDE。GLU _OUTSIDE 表示二次曲面的法线是自曲面向外的。GLU _INSIDE 则表示二次曲面的法线是指向二次曲面的。此时，球的诸法线均指向球心。其原型为：

*void gluQuadricOrientation(GLUquadricObj * quadObject, GLenum orientation);*

gluQuadricTexture()函数说明是否生成纹理坐标。它有两个参数，第一个参数是一个二次曲面对象指针。第二个参数为一个 OpenGL 常量，它说明是否为二次曲面生成纹理坐标，可以为 GL _TRUE 和 GL _FALSE。GL _TRUE 则表示生成纹理坐标，GL _FALSE 则表示不生成纹理坐标。

当用 gluNewQuadric()函数生成一个二次曲面对象之后，就占据了相应的内存。为了释放内存可以采用 gluDeleteQuadric()函数，比如：

gluDeleteQuadric(quadObj);

gluSphere()函数绘制一个球。该函数有 4 个参数。第一个参数是一个二次曲面对象的指针。第二个参数为球的半径。第三和第四个参数分别表示沿经线和纬线的线圈数。此时，球的半径为 0.7，经线和纬线数目均为 32。经线和纬线数目越多，组成球的平面越多，其表面就越光滑。下面我们来解释绘制其他二次曲面的函数。

gluCylinder()函数可以绘制圆柱或圆锥或台锥。该函数有 6 个参数。第一个参数是一个二次曲面对象的指针。第二和第三个参数分别是台锥体底面半径和顶面半径。第四个参数为台锥的高。第五和第六个参数分别相当于经线和纬线的线圈数。很显然，如果底面半径和顶面半径相等，则生成一个圆柱；如果底面半径和顶面半径中有一个为零，则生成一个圆锥体；如果二者不相等，则生成一个台锥。其函数原型为：

void gluCylinder(GLUquadricObj * qobj, GLdouble baseRadius,
GLdouble topRGLdouble height, GLint slices, GLint stacks);

gluDisk()函数绘制一个圆环或圆盘。该函数共有 5 个参数。第一个参数是一个二次曲面对象的指针。第二和第三个参数分别是圆环的内径和外径。第四和第五个参数分别表示径向的线圈数和周向的等分直线数目。当然如果内径为零，则生成一个圆盘。其原型为：

void gluDisk(GLUquadricObj * qobj,GLdouble innerRadius, GLdouble outerRadius,
GLint slices, GLint loops);

gluPartialDisk()函数绘制一个扇形。该函数共有7个参数。第一个参数是一个二次曲面对象的指针。第二和第三个参数分别是扇形的内径和外径。第四和第五个参数分别表示径向的线圈数和周向的等分直线数目。第六个参数表示起始角度。第七个参数表示扇形的张开角度。当然,只有当内径为零时,才会是一个真正的扇形。该函数的原型为:

void gluPartialDisk(GLUquadricObj * qobj, GLdouble innerRadius, GLdouble outerRadius,GLint slices, GLint loops, GLdouble startAngle, GLdouble sweepAngle);

12.1.2 绘制二次曲面

本节我们将绘制一个只画顶点的球、一个圆柱体、一个圆锥体、一个台锥、一个圆盘、一个圆环和一个扇形。请按以下步骤完成:

(1) 建立一个名为 Utility 的工程,并按第5章中的方法极小化 OpenGL。

(2) 添加枚举变量及保护成员变量

在 Utility View.h 文件的头部添加:

```
enum{Quadrics, ScaleDib,AutoMipmap};
```

增加一个保护成员变量:

```
GLint m_typeControl;
```

初始化为:

```
m_typeControl = Quadrics;
```

(3) 添加保护成员函数

添加一个保护成员函数:DrawQuadrics()。编辑如下:

```
void CUtilityView::DrawQuadrics()
{
    //光照与材质
    GLfloat position[] = { 0.0, 0.0, 3.0, 0.0 };
    glLightfv(GL_LIGHT0, GL_POSITION, position);

    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    {
        GLfloat material[3] = {0.2f, 0.05f, 0.05f};
        glMaterialfv (GL_FRONT, GL_AMBIENT, material);

        material[0] = 1.0f;
        material[1] = 0.2f;
        material[2] = 0.1f;
        glMaterialfv (GL_FRONT, GL_DIFFUSE, material);

        material[0] = 0.8f;
```

```
        material[1] = 0.8f;
        material[2] = 0.8f;
        glMaterialfv (GL_FRONT, GL_SPECULAR, material);
        glMaterialf (GL_FRONT, GL_SHININESS, 60.0);
    }
    GLUquadricObj * quadObj;

    //点球
    glNewList(1, GL_COMPILE);
        quadObj = gluNewQuadric();
        gluQuadricDrawStyle(quadObj,GLU_POINT);
        gluQuadricNormals(quadObj,GLU_SMOOTH);
        gluSphere(quadObj, 0.4,32,32);
    glEndList();

    //圆锥
    glNewList(2, GL_COMPILE);
        quadObj = gluNewQuadric();
        gluQuadricDrawStyle(quadObj,GLU_FILL);
        gluQuadricNormals(quadObj,GLU_SMOOTH);
        gluCylinder(quadObj,0.0,0.3,0.8,16,16);
    glEndList();

    //圆柱
    glNewList(3, GL_COMPILE);
        quadObj = gluNewQuadric();
        gluQuadricDrawStyle(quadObj,GLU_FILL);
        gluQuadricNormals(quadObj,GLU_SMOOTH);
        gluCylinder(quadObj,0.3,0.3,0.8,16,16);
    glEndList();

    //台锥
    glNewList(4, GL_COMPILE);
        quadObj = gluNewQuadric();
        gluQuadricDrawStyle(quadObj,GLU_FILL);
        gluQuadricNormals(quadObj,GLU_SMOOTH);
        gluCylinder(quadObj,0.1,0.3,0.6,16,16);
    glEndList();

    //圆环
    glNewList(5, GL_COMPILE);
        quadObj = gluNewQuadric();
        gluQuadricDrawStyle(quadObj,GLU_FILL);
        gluQuadricNormals(quadObj,GLU_SMOOTH);
```

```
        gluQuadricOrientation(quadObj,GLU_OUTSIDE);
        gluDisk(quadObj,0.1,0.3,16,16);
    glEndList();

    //圆盘
    glNewList(6, GL_COMPILE);
        quadObj = gluNewQuadric();
        gluQuadricDrawStyle(quadObj,GLU_FILL);
        gluQuadricNormals(quadObj,GLU_SMOOTH);
        gluDisk(quadObj,0.0,0.3,16,16);
    glEndList();

    //扇形
    glNewList(7, GL_COMPILE);
        quadObj = gluNewQuadric();
        gluQuadricDrawStyle(quadObj,GLU_FILL);
        gluQuadricNormals(quadObj,GLU_SMOOTH);
        gluPartialDisk(quadObj,0.1,0.3,16,16,0.0,60);
    glEndList();

    glPushMatrix();

        glTranslatef(-1.0f,0.8f,0.0f);
        glCallList(5);
        glTranslatef(0.7f,0.0f,0.0f);
        glCallList(6);
        glTranslatef(0.7f,0.0f,0.0f);
        glCallList(7);

        glTranslatef(-0.4f,-0.8f,0.0f);
        glRotatef(90.0f,1.0f,0.0f,0.0f);
        glCallList(2);
        glTranslatef(-1.0f,0.0f,0.0f);
        glCallList(1);
        glTranslatef(2.0f,0.0f,0.0f);
        glCallList(3);
        glTranslatef(0.0f,0.0f,-1.0f);
        glCallList(4);

    glPopMatrix();
}
```

一般来说,绘制二次曲面最好采用显示列表来管理。

(4) 添加菜单

添加一个名为 Type 的菜单,即在属性对话框的 Caption 栏内加入 &Quadrics。增加以下菜单项,下表是它们的属性对话框中的 Caption 及 ID 号。

表 12-1

Caption	ID
&Quadrics	IDM _ QUADRICS

用常用的方法响应消息 IDM _ QUADRICS:

```
void CUtilityView::OnQuadrics()
{
    // TODO: Add your command handler code here
    m_typeControl = Quadrics;
    Invalidate();
}

void CUtilityView::OnUpdateQuadrics(CCmdUI* pCmdUI)
{
    // TODO: Add your command update UI handler code here
    if(m_typeControl == Quadrics)
        pCmdUI->SetCheck(TRUE);
    else
        pCmdUI->SetCheck(FALSE);
}
```

(5) 编辑函数 DrawScene()

```
void CUtilityView::DrawScene()
{
    BeginWaitCursor();
    glClearColor(1.0f,1.0f,1.0f,1.0f);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glTranslatef(0.0f,0.0f,-4.5f);

    switch(m_typeControl)
    {
        case Quadrics:      DrawQuadrics();break;
    }

    glFinish();
    EndWaitCursor();
}
```

(6) 运行程序

编译连接并运行程序,结果在画面上生成了球、圆柱体、圆锥体、台锥、圆盘、圆环、扇形,如图 12-1 所示。

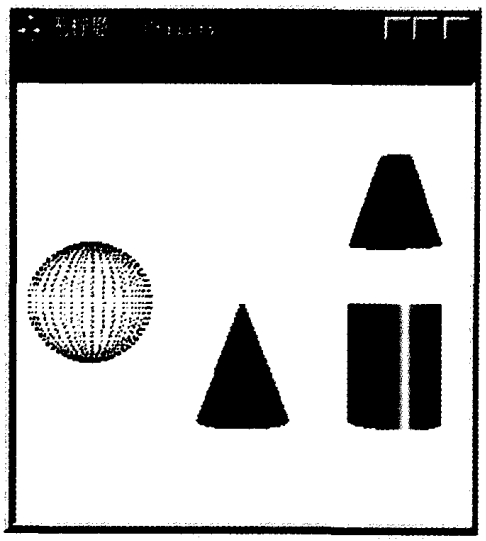

图 12-1 绘制二次曲面

12.2 图像处理

在第 9 章,我们用 glPixelZoom()函数来放大图像,在 OpenGL 实用库中,也有相应的放大图像的函数:gluScaleImage()。该函数可将一个图像放大到任意尺寸。不像 glPixelZoom()函数总影响后继图像,gluScaleImage()函数只影响它所处理的图像。其原型如下:

```
int gluScaleImage( GLenum format, GLint widthin, GLint heightin, GLenum typein,
                   const void * datain, GLint widthout, GLint heightout, GLenum
                   typeout, void * dataout );
```

其中,format 参数表示图像像素数据的类型;widthin,heightin 参数分别表示原图像的长和宽;typein 参数表示原图像素数据类型;datain 参数为指向原图像的地址指针;widthout,heightout 参数表示缩放后的图像的长和宽;typeout 参数表示缩放后的图像像素数据类型;dataout 参数表示缩放后的图像的地址指针。如果函数调用成功,则返回 0。

下面,我们应用该函数来缩放图像。

(1) 给工程添加外部文件

从 Texture 工程所在的目录拷贝文件 Dib.h、Dib.Cpp 和 Wheat.Bmp 等文件至 Utility 工程所在的目录。为 Utility 工程添加两个外部文件:Dib.h 和 Dib.Cpp。这相当于为该工程添加了一个处理位图的工具类:Cdib 类。

(2) 添加文件包含和公有成员变量

在文件 UtilityView.h 的头部添加文件包含:

```
# include "Dib.h"。
```

为 CUtilityView 类添加公有成员变量:

```
CDib * m_pDib;
```

(3) 添加保护成员函数

为 CUtilityView 类添加一个保护成员函数:ScaleImage()。该函数读入一个位图至内存,并用 gluScaleImage()函数对其放大 4 倍,然后显示在用户区。编辑如下:

```
void CUtilityView::ScaleImage()
{
    m_pDib = new CDib("Wheat.BMP");

    GLubyte * pDibBits = (GLubyte * )m_pDib->GetDibBitsPtr();

    GLint width = m_pDib->GetDibWidth();
    GLint height = m_pDib->GetDibHeight();

    glPixelStorei(GL_UNPACK_ALIGNMENT,1);

    GLubyte * image;
    image = new GLubyte[width * height * 12];

    ASSERT(image != NULL);

    gluScaleImage(GL_BGR_EXT,width,height,GL_UNSIGNED_BYTE,pDibBits,
                  2 * width,2 * height,GL_UNSIGNED_BYTE,image);

    glRasterPos3f(-1.5f,-1.5f,0.0f);
    glDrawPixels(2 * width,2 * height,GL_BGR_EXT,GL_UNSIGNED_BYTE,image);
    delete []image;
}
```

(4) 添加菜单

添加一个名为 Image 的菜单,即在属性对话框的 Caption 栏内加入 &Image。增加以下菜单项,表 12-2 是它们的属性对话框中的 Caption 及 ID 号。

表 12-2

Caption	ID
Sc&ale Image	IDM_SCALE_IMAGE
A&uto Mipmapping	IDM_AUTO_MIPMAPPING

按响应消息 IDM_QUADRICS 相应的方法响应上述诸消息:

```
void CUtilityView::OnScaleImage()
{
    // TODO: Add your command handler code here
```

```
    m_typeControl = ScaleDib;
    Invalidate();
}

void CUtilityView::OnUpdateScaleImage(CCmdUI* pCmdUI)
{
    // TODO: Add your command update UI handler code here
    if(m_typeControl == ScaleDib)
        pCmdUI->SetCheck(TRUE);
    else
        pCmdUI->SetCheck(FALSE);
}

void CUtilityView::OnAutoMipmapping()
{
    // TODO: Add your command handler code here
    m_typeControl = AutoMipmap;
    Invalidate();
}

void CUtilityView::OnUpdateAutoMipmapping(CCmdUI* pCmdUI)
{
    // TODO: Add your command update UI handler code here
    if(m_typeControl == AutoMipmap)
        pCmdUI->SetCheck(TRUE);
    else
        pCmdUI->SetCheck(FALSE);
}
```

(5) 编辑 DrawScene()函数

在结构 switch(){... ....}中添加语句:

```
case ScaleDib: ScaleImage(); break;
```

(6) 运行程序

重新编译连接并运行程序,并执行菜单 Image \ Scale Image,最后将原位图放大 4 倍。如图 12-2 所示。

12.3 自动 Mip 贴图

OpenGL 实用库提供了自动 Mip 贴图过程。利用该过程,不需要对纹理进行控制,所有控制都是自动完成的。自动 Mip 贴图过程首先获取输入图像,将图像缩放到合法的纹理大小,最后生成 Mip 贴图纹理。该过程由 gluBuild2DMipmaps()函数或 gluBuild1DMipmaps()函数完成。gluBuild2DMipmaps()函数的原型为:

图 12-2　放大图像

nt gluBuild2DMipmaps(GLenum target, GLint components, GLint width, GLint height,
GLenum format, GLenum type, const void * data);

其中,target 参数必为 GL _ TEXTURE _ 2D;components 参数为颜色分量的个数,可以为 1,2,3,4;width,height 参数为原图像长度和宽度;format 参数为图像像素数据格式;type 参数为像素数据类型;data 参数为图像存储地址。如果被成功调用就返回零,否则返回一个 GLU 错误。gluBuild1DMipmaps()函数的形式与 gluBuild2DMipmaps()函数的形式基本一样。

自动 Mip 贴图可以根据图像的大小和景深远近自动对图像进行缩放以适应物体的大小,并在一个图像和另一个图像相接处进行平滑处理。当对一个图像制作成 Mip 贴图纹理之后,还要用 glTexImage2D()函数将纹理数据传递给 OpenGL。

下面的例程是对第 10 章中的 Mip 贴图的一个修改,主要体现自动控制过程。

(1) 添加保护成员变量及函数

为 CUtilityView 类添加保护成员变量:

```
GLint      m _ iWindowWidth;
GLint      m _ iWindowHeight;
```

并初始化为:

```
m _ iWindowWidth = 400;
m _ iWindowHeight = 435;
```

注意,以上数据来源于 CMainFrame 类的成员函数 PreCreateWindow()中的语句:

```
cs.cx = 400;
cs.cy = 435;
```

如果 PreCreateWindow()函数中还没有这两条语句,请添加进去,如有请进行改动。这两个成员变量用来获取窗口的大小。

为 CUtilityView 类添加一个保护成员函数:AutoMipmapping(),并编辑如下:

```
void CUtilityView::AutoMipmapping()
{
    int i,j;
    GLubyte mipmap32[32][32][4];
    GLubyte mipmap16[16][16][4];
    GLubyte mipmap8[8][8][4];
    GLubyte mipmap4[4][4][4];
    GLubyte mipmap2[2][2][4];
    GLubyte mipmap1[1][1][4];

    //作 Mip 贴图的图像数据
    for (i = 0; i < 32; i++) {
    for (j = 0; j < 32; j++) {
      mipmap32[i][j][0] = 50;
      mipmap32[i][j][1] = 125;
      mipmap32[i][j][2] = 255;
        mipmap32[i][j][3] = 255;
    }
    }
    for (i = 0; i < 16; i++) {
    for (j = 0; j < 16; j++) {
      mipmap16[i][j][0] = 200;
      mipmap16[i][j][1] = 200;
      mipmap16[i][j][2] = 0;
        mipmap16[i][j][3] = 255;
    }
    }
    for (i = 0; i < 8; i++) {
    for (j = 0; j < 8; j++) {
      mipmap8[i][j][0] = 255;
      mipmap8[i][j][1] = 0;
      mipmap8[i][j][2] = 0;
        mipmap8[i][j][3] = 255;
    }
    }
    for (i = 0; i < 4; i++) {
    for (j = 0; j < 4; j++) {
      mipmap4[i][j][0] = 0;
      mipmap4[i][j][1] = 200;
      mipmap4[i][j][2] = 0;
        mipmap4[i][j][3] = 255;
    }
    }
    for (i = 0; i < 2; i++) {
```

```
for (j = 0; j < 2; j++) {
  mipmap2[i][j][0] = 0;
  mipmap2[i][j][1] = 0;
  mipmap2[i][j][2] = 200;
    mipmap2[i][j][3] = 255;
}
}
mipmap1[0][0][0] = 255;
mipmap1[0][0][1] = 255;
mipmap1[0][0][2] = 255;
mipmap1[0][0][3] = 255;

glEnable(GL_DEPTH_TEST);
glDepthFunc(GL_LESS);
glClearColor(0.0f,0.0f,0.0f,1.0f);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glShadeModel(GL_FLAT);
glPixelStorei(GL_UNPACK_ALIGNMENT, 1);

//第零级
gluBuild2DMipmaps(GL_TEXTURE_2D,GL_RGBA,32,32,GL_RGBA,
                  GL_UNSIGNED_BYTE,mipmap32);
//第一级
gluBuild2DMipmaps(GL_TEXTURE_2D,GL_RGBA,16,16,GL_RGBA,
                  GL_UNSIGNED_BYTE,mipmap16);
//第二级
gluBuild2DMipmaps(GL_TEXTURE_2D,GL_RGBA,8,8,GL_RGBA,
                  GL_UNSIGNED_BYTE,mipmap8);
//第三级
gluBuild2DMipmaps(GL_TEXTURE_2D,GL_RGBA,4,4,GL_RGBA,
                  GL_UNSIGNED_BYTE,mipmap4);
//第四级
gluBuild2DMipmaps(GL_TEXTURE_2D,GL_RGBA,2,2,GL_RGBA,
                  GL_UNSIGNED_BYTE,mipmap2);
//第五级
gluBuild2DMipmaps(GL_TEXTURE_2D,GL_RGBA,1,1,GL_RGBA,
                  GL_UNSIGNED_BYTE,mipmap1);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, 32, 32, 0,
             GL_RGBA, GL_UNSIGNED_BYTE, &mipmap32);
glTexImage2D(GL_TEXTURE_2D, 1, GL_RGBA, 16, 16, 0,
             GL_RGBA, GL_UNSIGNED_BYTE, &mipmap16);
glTexImage2D(GL_TEXTURE_2D, 2, GL_RGBA, 8, 8, 0,
             GL_RGBA, GL_UNSIGNED_BYTE, &mipmap8);
glTexImage2D(GL_TEXTURE_2D, 3, GL_RGBA, 4, 4, 0,
```

```
                GL_RGBA, GL_UNSIGNED_BYTE, &mipmap4);
    glTexImage2D(GL_TEXTURE_2D, 4, GL_RGBA, 2, 2, 0,
                GL_RGBA, GL_UNSIGNED_BYTE, &mipmap2);
    glTexImage2D(GL_TEXTURE_2D, 5, GL_RGBA, 1, 1, 0,
                GL_RGBA, GL_UNSIGNED_BYTE, &mipmap1);
    glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_DECAL);

    glEnable(GL_TEXTURE_2D);
    //放大视线,体现 Mip 贴图的作用
    gluPerspective(60.0, 1.0*(GLfloat)m_iWindowWidth/(GLfloat)m_iWindowHeight,
                   1.0, 30000.0);
    //Mip 贴图处理的物体
    glBegin(GL_QUADS);
        glTexCoord2f(0.0, 0.0); glVertex3f(-2.0, -1.0, 0.0);
        glTexCoord2f(0.0, 5.0); glVertex3f(-2.0, 1.0, 0.0);
        glTexCoord2f(5.0, 5.0); glVertex3f(1500.0, 1.0, -5000.0);
        glTexCoord2f(5.0, 0.0); glVertex3f(1500.0, -1.0, -5000.0);
    glEnd();

    glDisable(GL_TEXTURE_2D);
    glShadeModel(GL_SMOOTH);
    //视线回位
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glFrustum(-1.0,1.0,-1.0,1.0,3.0,7.0);
}
```

(2) 编辑 OnSize()函数

为了时实地获取窗口大小变化的信息以便图像在原来的长宽比上呈比例地缩放,必须把长宽变比信息传递给变量 m_iWindowWidth 和变量 m_iWindowHeight。编辑 OnSize()函数:

```
void CUtilityView::OnSize(UINT nType, int cx, int cy)
{
    CView::OnSize(nType, cx, cy);
    // TODO: Add your message handler code here

    m_iWindowWidth = cx;
    m_iWindowHeight = cy;

     if(cy > 0)
    {
        if((m_oldRect.right > cx) || (m_oldRect.bottom > cy))
        RedrawWindow();
      m_oldRect.right = cx;
```

```
        m_oldRect.bottom = cy;
        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
          glFrustum(-1.0,1.0,-1.0,1.0,3.0,7.0);
          glViewport(0, 0, cx, cy);
    }
}
```

(3) 编辑 DrawScene()函数

在 switch(m_typeControl){.....}中添加如下语句:

```
case AutoMipmap: AutoMipmapping(); break;
```

(4) 运行程序

再次编译连接并运行程序,执行菜单 Image \ Auto Mipmapping,结果 OpenGL 进行了自动 Mip 贴图,而且对图像与图像的相接部位进行了平滑过渡处理,如图 12-3 所示。Mip 贴图对较大场景的贴图既能保证足够的精度又能提高着色效率。

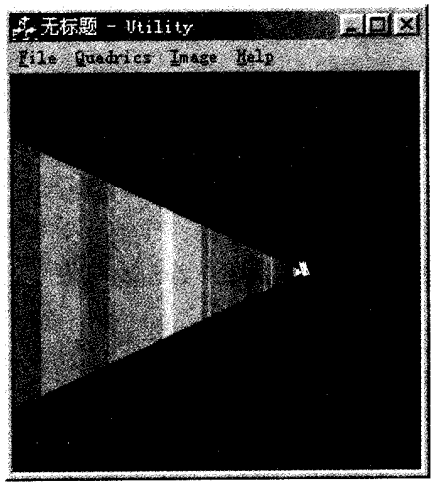

图 12-3 自动 Mip 贴图

第13章　曲线和曲面

13.1　概　述

曲线和曲面是计算机图形学中研究的重要内容之一，它们在实际工作中有着广泛的应用。例如，实验数据的曲线表示，设计、优化的曲面表示等。为了外形美观和物理性能最佳，汽车、飞机等的外形设计十分重要。由于实际问题不断对曲线和曲面提出新的要求，近几十年来，曲线和曲面理论及其应用得到很大的发展。1963年，波音飞机公司的Ferguson将曲线曲面表示成参数矢量函数形式，并用3次参数曲线来构造组合曲线，用4个角点的位置矢量及其两个方向的切向矢量来构造3次曲面。1964年，麻省理工学院的Coons用封闭曲线的4条边界定义一个曲面。同年，Schoenberg给出了参数样条曲线和曲面的形式。1971年，法国雷诺汽车公司的Bezier提出了用控制多边形定义曲线和曲面的方法。也在同一年，法国雪铁龙汽车公司的De Casteljau独立地研究出与Bezier类似的方法。1972年，De Boor给出了B样条的标准计算方法。1974年，美国通用汽车公司的Gordon和Riesenfeld将B样条用于形状描述，提出了B样条曲线和B样条曲面。1975年，美国锡拉丘兹(Syracuse)大学的Versprill在其博士论文中提出了有理B样条方法。19世纪80年代后期，美国的Piegl和Tiller将有理B样条发展成为非均匀有理B样条(Non-Uniform Rational B_Spline，即NURBS)方法。非均匀有理B样条方法已成为当今自由曲线和曲面描述的通用方法，可以统一表示初等解析曲线和曲面、有理与非有理Bezier曲线和曲面以及有理与非有理B样条曲线和曲面。

13.2　Bezier曲线

13.2.1　Bezier曲线

Bezier曲线是一种以逼近为基础的参数曲线，它是由一组Bezier特征多边形来定义的。曲线的起点和终点与该多边形的起点和终点重合，且多边形的第一条边和最后一条边分别表示了曲线在起点和终点处的切向矢量方向。曲线的形状则趋于多边形的形状。多边形可由其顶点来定义，这些顶点则被称为控制点。只要给出控制点，就可生成一条Bezier曲线。

在OpenGL中，曲线和曲面的构造是借助于OpenGL评价器来完成的。要生成一条曲线，首先要创建一个一维评价器(也称为求值器)。评价器是基于OpenGL而建立的一个生成更一般曲线和曲面包的工具。利用评价器可自动生成顶点坐标、法线坐标和纹理坐标。其次要激活评价器，使其进行曲线映射。最后要将评价器生成的诸顶点连接起来以生成一条完整的曲线。

创建一维评价器的函数为：glMap1d()或glMap1f()函数，由它们生成所需坐标值。以glMap1d()函数为例，其原型为：

```
void glMap1d( GLenum target, GLdouble u1, GLdouble u2, GLint stride, GLint order,
              const GLdouble * points );
```

其中，target 参数是一个标识参数，它表示控制点应该组织成什么样的数据形式，以及当评价器被成功调用后输出数据的形式。它可以取表 13-1 中的任意一个值。

表 13-1

常　量	含　义
GL_MAP1_VERTEX_3	用(x,y,z)描述一个控制点
GL_MAP1_VERTEX_4	用(x,y,z,w)描述一个控制点
GL_MAP1_INDEX	控制点代表一个颜色索引值
GL_MAP1_COLOR_4	控制点是 RGBA 颜色值
GL_MAP1_NORMAL	控制点是一个法线向量
GL_MAP1_TEXTURE_COORD_1	控制点是一个纹理坐标的 s 分量
GL_MAP1_TEXTURE_COORD_2	控制点是一个(s,t)纹理坐标
GL_MAP1_TEXTURE_COORD_3	控制点是一个(s,t,r)纹理坐标
GL_MAP1_TEXTURE_COORD_4	控制点是一个(s,t,r,q)纹理坐标

u1 和 u2 参数表示调和函数的变量 u 的取值范围；stride 参数表示控制点向量的维数，可以与 target 参数表示的含义不一致；order 参数为控制点的个数；points 参数为控制点地址指针。

创建一个一维曲线评价器之后，就应该激活评价器使其进入工作状态。由下述语句完成：

```
glEnable(GL_MAP1_VERTEX_3);
```

注意，glEnable()函数的参数应该与 glMap1d()函数的第一个参数一致。当不再需要进行映射之后，相应地，就应该挂起评价器：

```
glEnable(GL_MAP1_VERTEX_3);
```

为了能生成一条曲线，最后还要进行曲线坐标的计算和连接。该函数为：glEvalCoord1d()或 glEvalCoord1f()。以 glEvalCoord1d()为例，其原型为：

```
void glEvalCoord1d( GLdouble u );
```

其中，u 参数表示参数空间 UVW 中 u 参数的取值。给定一个 u 值，就会产生一个曲线坐标。当用 glEvalCoord1()函数生成曲线坐标之后，还要将这些坐标连接起来构成一条曲线。这可以通过 glBegin()/glEnd()函数对来完成。

OpenGL 允许对 u 参数区间进行自动等分，以获得等间距的曲线上的点。该函数为：glMapGrid1d()或 glMapGrid1f()。以 glMapGrid1d()函数为例其原型为：

```
void glMapGrid1d( GLint un, GLdouble u1, GLdouble u2 );
```

其中，un 参数表示参数区间等分数；u1，u2 参数表示参数区间[u1,u2]，必须均为非负值。当对参数区间进行自动等分之后，还需要用 glEvalMesh1()函数产生一系列的点并利用这些顶点坐标最终绘制一条曲线。该函数的原型为：

void glEvalMesh1(GLenum mode, GLint i1, GLint i2);

其中,mode 参数可以为 GL_POINT 或 GL_LINE,它们分别表示将曲线绘制成点还是绘制成线;i1,i2 参数表示只绘制从第 i1 个点到第 i2 个点相对应的这段区间上的曲线。

13.2.2 绘制一条拟正弦曲线

下面,我们绘制一条看起来像正弦线的 Bezier 曲线。为了生成一条这样的曲线,必须正确地提供控制点及其走向。曲线的起点和终点是要经过控制点的,但其他的点则不一定通过控制点。在绘制这条曲线时,我们将采用两种方法,第一种方法是借助于 glBegin()/glEnd()函数对来完成曲线的绘制,第二种方法是自动等分参数区间并进行绘制。请按如下步骤进行:

(1) 建立一个名为 CuvSufc 的工程,并按第 5 章中的方法极小化 OpenGL。

(2) 添加枚举变量及保护成员变量

在 CuvSufc View.h 文件的头部添加:

```
enum{ BezierCurve,          EvenBezier,          NurbsCurve,
      BezierSurfaceMesh,    BezierSurfaceFill,   BezierSurfaceTex,
      NurbsSurfaceMesh,     NurbsSurfaceFill,    NurbsSurfacePach
      NurbsSurfaceTrim};
```

增加一个保护成员变量:

```
GLint m_typeControl;
```

并初始化为:

```
m_typeControl = BezierCurve;
```

该变量控制显示类别。

(3) 添加保护成员函数

添加两个保护成员函数:DrawBezierCurve()和 EvenBezierCurve()。DrawBezierCurve()函数按第一种方式绘制,EvenBezierCurve()按第二种方式绘制,现分别编辑如下:

```
void CCuvSufcView::DrawBezierCurve()
{
    glColor3f(1.0, 0.0, 0.0);
    //控制点
    GLfloat controlPoints[9][3] = {
        { -1.0f, 0.0f, 0.0f},    { -0.75f, 0.707f, 0.0f},    { -0.5f, 1.0f, 0.0f},
        { -0.25f, 0.707f, 0.0f}, {0.0f,0.0f,0.0f},           { 0.25f, -0.707f, 0.0f},
        { 0.5f, -1.0f, 0.0f},    { 0.75f, -0.707f, 0.0f},    {1.0f, 0.0f, 0.0f}
    };
    //一维评价器
    glMap1f(GL_MAP1_VERTEX_3, 0.0, 1.0, 3, 9,(const float*) controlPoints);
    //激活
    glEnable(GL_MAP1_VERTEX_3);
    //计算曲线坐标并连接
```

```
    glBegin(GL_LINE_STRIP);
        for (int i = 0; i <= 50; i++)
            glEvalCoord1f((GLfloat) i/(GLfloat)50.0);
    glEnd();
    //挂起
    glDisable(GL_MAP1_VERTEX_3);
}
```

要注意,当用评价器获得曲线的近似点以后,还需要用直线段的方法将它们连接成线的形式,glMap1f()函数并不能完成这一工作。因而需要执行如下语句:

```
    glBegin(GL_LINE_STRIP);
        for (int i = 0; i <= 50; i++)
            glEvalCoord1f((GLfloat) i/(GLfloat)50.0);
    glEnd();
```

其中,数值 50 可以理解为共用 50 段来逼近这条曲线,该数值越高,逼近程度越好,但所需要的计算将越多。

```
void CCuvSufcView::EvenBezierCurve()
{
    glColor3f(0.0f, 0.0f, 1.0f);
    //控制点
    GLfloat controlPoints[9][3] = {
        {-1.0f, 0.0f, 0.0f},        {-0.75f, 0.707f, 0.0f},      {-0.5f, 1.0f, 0.0f},
        {-0.25f, 0.707f, 0.0f},     {0.0f,0.0f,0.0f},            {0.25f, -0.707f, 0.0f},
        {0.5f, -1.0f, 0.0f},        {0.75f, -0.707f, 0.0f},      {1.0f, 0.0f, 0.0f}
    };
    glMap1f(GL_MAP1_VERTEX_3, 0.0, 1.0, 3, 9,(const float*)controlPoints);
    glEnable(GL_MAP1_VERTEX_3);
    //自动等分区间
    glMapGrid1f(50,0.0f,1.0f);
    //绘制曲线
    glEvalMesh1(GL_LINE,0,50);
    glDisable(GL_MAP1_VERTEX_3);
}
```

该函数与 DrawBezierCurve()函数的重要区别在于应用了 glMapGrid1f()和 glEvalMesh1()函数,glMapGrid1f()函数将一维参数空间等分为 50 份,由 glEvalMesh1()函数来完成这 50 份的连接。两函数的数值为对应的 50,当然也可以不进行匹配。比如:以下语句将最终输出参数空间[0.0,0.5]对应的曲线:

```
    glEvalMesh1(GL_LINE,0,25);
```

以上两个自定义函数绘制一条类似正弦曲线的曲线,均由 9 个控制点来决定。这 9 个控制点的走向是沿 y = sin(x)曲线的,最终绘制的曲线却不是 y = sin(x)曲线,因为,它只通过其

端点和中点,但看起来像一条正弦曲线。如何绘制标准正弦曲线和标准二次曲线?首先应将它们转化为 Bezier 曲线表示,然后再构造控制点,将控制点信息输入至评价器,让其进行计算,最后决定采用何种形式输出曲线。相关算法请参考有关计算机图形学的资料。

(4) 添加菜单

添加一个名为 Curve 的菜单,即在属性对话框的 Caption 栏内加入 C&urve。增加以下菜单项,表 13-2 是它们的属性对话框中的 Caption 及 ID 号。

表 13-2

Caption	ID
&Bezier	IDM_BEZIER_CURVE
&Even Bezier	IDM_EVEN_BEZIER_CUV
&Nurbs	IDM_NURBS_CURVE

用常用的方法响应上述诸消息:

```
void CCuvSufcView::OnBezierCurve()
{
    // TODO: Add your command handler code here
    m_typeControl = BezierCurve;
    Invalidate();
}

void CCuvSufcView::OnUpdateBezierCurve(CCmdUI* pCmdUI)
{
    // TODO: Add your command update UI handler code here
    if(m_typeControl == = BezierCurve)
        pCmdUI->SetCheck(TRUE);
    else
        pCmdUI->SetCheck(FALSE);
}

void CCuvSufcView::OnEvenBezierCuv()
{
    // TODO: Add your command handler code here
    m_typeControl = EvenBezier;
    Invalidate();
}

void CCuvSufcView::OnUpdateEvenBezierCuv(CCmdUI* pCmdUI)
{
    // TODO: Add your command update UI handler code here
    if(m_typeControl == = EvenBezier)
        pCmdUI->SetCheck(TRUE);
```

```
    else
        pCmdUI->SetCheck(FALSE);
}

void CCuvSufcView::OnNurbsCurve()
{
    m_typeControl=NurbsCurve;
    Invalidate();
}

void CCuvSufcView::OnUpdateNurbsCurve(CCmdUI* pCmdUI)
{
    if(m_typeControl==NurbsCurve)
        pCmdUI->SetCheck(TRUE);
    else
        pCmdUI->SetCheck(FALSE);
}
```

(5) 编辑 DrawScene()函数

```
void CCuvSufcView::DrawScene()
{
    glClearColor(1.0f,1.0f,1.0f,1.0f);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glTranslatef(0.0f,0.0f,-4.5f);
    switch(m_typeControl)
    {
        case BezierCurve: DrawBezierCurve(); break;
        case EvenBezier: EvenBezierCurve(); break;
    }
    glFinish();
}
```

(6) 运行程序

编译连接并运行程序,执行菜单 Curve \ Bezier 和 Curve \ Even Bezier。结果绘制一条拟正弦曲线。为了区别,一条为红色,另一条则为蓝色。如图 13-1 所示。

13.3 NURBS 曲线

13.3.1 NURBS 曲线的生成

BURBS 曲线是对均匀有理 B 样条曲线的改进。不像均匀 B 样条函数,非均匀 B 样条函

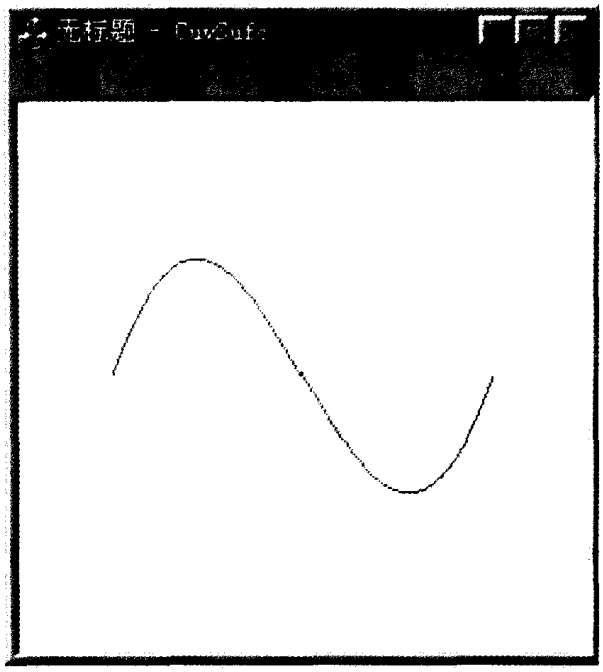

图 13-1　Bezier 曲线

数其节点参数沿参数轴的分布是不均匀的，因而不同节点矢量形成的 B 样条函数各不相同，这就增大了控制的灵活性。工程上，NURBS 有十分广泛的应用，主要是：

第一，对标准的解析形状（如圆锥曲线、二次曲面、回转面等）和自由曲线、曲面提供了统一的数学表示，便于工程数据库的存取和应用；

第二，可通过控制点和权因子来灵活地改变形状；

第三，对插入节点、分割、几何插值和修改等的处理比较容易控制；

第四，具有透视投影变换和仿射变换不变性。

但是，在应用 NURBS 曲线时，还有一些需要处理的问题：

第一，NURBS 曲线（包括 NURBS 曲面）比一般的曲线、曲面定义方法更需要存储空间和处理时间。

第二，构造 NURBS 曲线（曲面）时，需要确定权因子，如果选择不当会造成形状畸形（注意，OpenGL 保留权因子的选择）。这是构造新曲线和曲面以及移植已有曲线和曲面时必须解决的问题。

第三，对搭接、重叠形状的处理相当麻烦。

第四，当需要通过点的映射算法来为 NURBS 曲线（曲面）获得计算信息时，NURBS 算法有可能不够稳定。

在 OpenGL 中，GLU 函数库提供了一个 NURBS 接口，该接口连接 OpenGL 评价器，并通过评价器连接 OpenGL 的内核，最终实现用户交互。用户需要提供的重要数据包括控制点、节点和纹理等数据。控制点说明曲线的大致走向。而节点则控制 B 样条函数的形状，因而最终控制曲线的形状。绘制一个 NURBS 曲线至少要完成如下步骤：

第一，提供控制点序列和节点序列；

第二，创建一个 NURBS 对象，设置 NURBS 对象属性；

第三，绘制曲线。

要创建一个 NURBS 对象，首先应定义一个 NURBS 对象指针，然后用 gluNewNurbsRenderer()函数来创建，如以下两条语句：

```
GLUnurbsObj * theNurb;
theNurb = gluNewNurbsRenderer();
```

在此之后，应设置 NURBS 对象的属性，通过 gluNurbsProperty()函数来完成。该函数的原型为：

void gluNurbsProperty(GLUnurbsObj * nobj, GLenum property, GLfloat value);

nobj 参数为一个 NURBS 对象指针，由 gluNewNurbsRenderer()函数创建。property 参数可取表 13-3 中的值。

value 参数说明属性的值。其取值较复杂，它可以为一个浮点数，也可以为一个常量，主要由 property 参数来决定。当 property 参数为 GLU _ DISPLAY _ MODE 时，value 可以取如下三者之一：GLU _ FILL、GLU _ OUTLINE _ POLYGON 或 GLU _ OUTLINE _ PATCH。其含义见表 13-4。若 property 参数为 GLU _ SAMPLING _ METHOD，value 的取值可参见表 13-5。

表 13-3

常 量	含 义
GLU _ SAMPLING _ TOLERANCE	边缘的最大像素长度
GLU _ DISPLAY _ MODE	绘制曲线曲面的模式
GLU _ CULLING	细化时如何处理 NURBS 曲线
GLU _ AUTO _ LOAD _ MATRIX	自动下载变换矩阵
GLU _ PARAMETRIC _ TOLERANCE	最大绘制步长
GLU _ SAMPLING _ METHOD	怎样细化一个 NURBS 曲面
GLU _ U _ STEP	参数 u 单位长度的取样步长
GLU _ V _ STEP	参数 v 单位长度的取样步长

表 13-4

常 量	含 义
GLU _ FILL	绘制一张填充曲面
GLU _ OUTLINE _ POLYGON	只绘制细化多边形的轮廓
GLU _ OUTLINE _ PATCH	绘制用户定义的曲面小片和修剪回路

表 13-5

常 量	含 义
GLU _ PATH _ LENGTH	细化多边形的边缘长度不超过由 GLU _ SAMPLING _ TOLERANCE 指定的值
GLU _ PARAMETRIC _ ERROR	说明细化多边形与曲面间的误差参考由 LU _ PARAMETRIC _ TOLERANCE 指定的值
GLU _ DOMAIN _ DISTANCE	参数空间的取样距离

最后，应根据所提供的外部数据(如控制点列、节点序列、法线矢量数据、纹理数据等)绘制曲线。

与二维形状的绘制过程一样，曲线的绘制是在 gluBeginCurve()/gluEndCurve()函数对中完成的。这两个函数的参数都是一个 NURBS 对象。绘制曲线的函数为：gluNurbsCurve()，其原型为：

```
void gluNurbsCurve( GLUnurbsObj * nobj, GLint nknots, GLfloat * knot,
                    GLint stride, GLfloat * ctlarray, GLint order, GLenum type );
```

其中，nobj 参数是一个 NURBS 对象；nknots 参数表示 u 参数区间的节点数目，它等于控制点个数加上 NURBS 曲线的阶数；knot 参数表示节点序列的指针；stride 参数表示一个控制点的分量个数；ctlarray 参数表示控制点序列的指针；order 参数表示 NURBS 曲线的阶数；type 参数表示曲线的类型，可以取 GL_MAP1_VERTEX_3、GL_MAP1_COLOR_4、GL_MAP1_TEXTURE_COORD_2、GL_MAP1_NORMAL 等值。

13.3.2 绘制一条 NURBS 曲线

请按下列步骤绘制一条 NURBS 曲线：

(1) 添加保护成员函数

添加一个保护成员函数：DrawNurbsCurve()，并编辑如下：

```
void CCuvSufcView::DrawNurbsCurve()
{
    //控制点
    GLfloat controlPoints[7][3] = {
        {-1.2f, 0.8f, 0.0f},    {-0.8f, -2.0f, 0.0f},    {-0.4f, 0.4f, 0.0f},
        {0.0f, 2.0f, 0.0f},     {0.4f, 0.4f, 0.0f},      {0.8f, -2.0f, 0.0f},
        {1.2f, 0.8f, 0.0f}
      };
    //节点
    GLfloat knots[14] = {
        0.0f,    0.0f,    0.0f,    0.0f,    0.0f,    0.0f,
        1.0f,    1.0f,    1.0f,    1.0f,    1.0f,    1.0f,
                                                      }
    glColor3f (0.0f, 0.0f, 1.0f);
    //创建 NURBS 对象
    GLUnurbsObj        * theNurb;
theNurb = gluNewNurbsRenderer();
    //属性
    gluNurbsProperty (theNurb, GLU_SAMPLING_TOLERANCE, 10.0);
    //绘制
    glNewList(1, GL_COMPILE);
        gluBeginCurve(theNurb);
            gluNurbsCurve(theNurb,
                14,
```

```
                knots,
                3,
                (float * )controlPoints,
                7,
                GL _ MAP1 _ VERTEX _ 3);
        gluEndCurve(theNurb);
    glEndList();
    glCallList(1);
```

(2) 编辑 DrawScene()函数

在语句 switch(m _ typeControl){.....}中添加语句:

```
case NurbsCurve:    DrawNurbsCurve();    break;
```

(3) 运行程序

再次编译连接并运行程序,执行菜单 Curve \ Nurbs,就可看见,一条曲线已经画出,如图 13-2所示。

必须指出,如果 gluNurbsCurve()函数变为如下形式的调用,则所提供和返回的信息将变成曲线颜色信息:

```
gluNurbsCurve(theNurb,16, knots,4,(float * )controlPoints, 8, GL _ MAP1 _ COLOR _ 4);
```

如果为以下调用形式,则为制作连通曲面返回边界曲线,且曲线是三维的空间曲线:

```
gluNurbsCurve(theNurb,14, knots,3,(float * )controlPoints, 7, GLU _ MAP1 _ TRIM _ 3);
```

如果为以下调用形式,则为制作连通曲面返回边界曲线,则曲线是二维的平面曲线:

```
gluNurbsCurve(theNurb,14, knots,3,(float * )controlPoints, 7, GLU _ MAP1 _ TRIM _ 2);
```

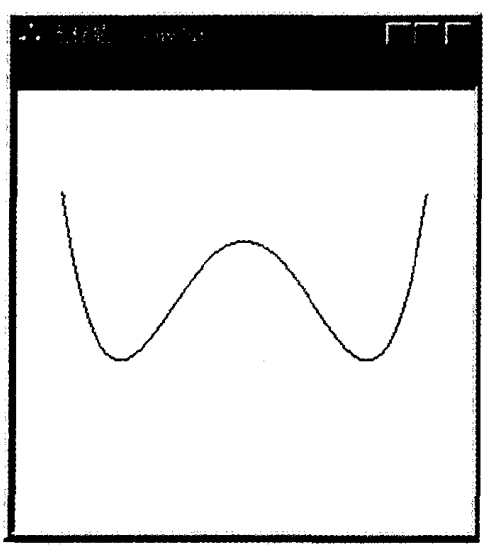

图 13-2 绘制 NURBS 曲线

13.4 Bezier 曲面

13.4.1 Bezier 曲面的生成

Bezier 曲面是基于 Bezier 曲线而定义的，由特征网格决定其大致形状，而特征网格是由控制点确定的。因而，绘制 Bezier 曲面的步骤与绘制 Bezier 曲线的步骤是一致的：

第一，定义控制点序列；

第二，定义一个二维评价器；

第三，激活二维评价器；

最后，创建网格。

定义二维评价器的函数为：glMap2d()或 glMap2f()函数，glMap2d()函数的原型为：

```
void glMap2d( GLenum target,    GLdouble u1,   GLdouble u2,   GLint ustride,
              GLint uorder,     GLdouble v1,   GLdouble v2,   GLint vorder,
              GLint uorder,     const GLdouble * points );
```

其中，target 参数表示评价器最后生成的坐标类别，它可以取表 13-6 中的一个常量：

表 13-6

常　量	含　义
GL_MAP2_VERTEX_3	用(x,y,z)描述一个控制点
GL_MAP2_VERTEX_4	用(x,y,z,w)描述一个控制点
GL_MAP2_INDEX	控制点代表一个颜色索引值
GL_MAP2_COLOR_4	控制点是 RGBA 颜色值
GL_MAP2_NORMAL	控制点是一个法线向量
GL_MAP2_TEXTURE_COORD_1	控制点是一个纹理坐标的 s 分量
GL_MAP2_TEXTURE_COORD_2	控制点是一个(s,t)纹理坐标
GL_MAP2_TEXTURE_COORD_3	控制点是一个(s,t,r)纹理坐标
GL_MAP2_TEXTURE_COORD_4	控制点是一个(s,t,r,q)纹理坐标

u1，u2 参数表示 u 参数的取值范围；ustride 参数表示在 u 方向上，上一个控制点与下一个控制点间的浮点数个数；uorder 参数表示 u 轴方向控制点矩阵的大小；v1、v2、vstride 及 vorder 参数则表示 v 轴方向上相应的量；points 参数表示控制点地址。

创建网格过程包括首先获取二维评价器的坐标计算结果，然后把几何坐标表示的点要么以点的形式输出，要么连接成线的形式输出，要么填充成多边形后输出。获取评价器的计算结果是由 glEvalCoord2f()函数完成的。下面我们通过一个例程来学习和应用它们。

13.4.2 绘制网格曲面

本节绘制一个网格曲面，请按如下步骤完成：

(1) 添加一个菜单

添加一个名为 Surface 的菜单,在其属性对话框的 Caption 栏中键入 Su&rface。为其增加一个菜单项,其名为 Bezier,在其属性对话框的 Caption 栏中键入 Bezier,并勾选 Pop-up 属性。现在为该菜单项添加 3 个子项,按表 13-7 完成。

表 13-7

Caption	ID
&Mesh	IDM_BZRSUFAC_MESH
&Fill	IDM_BZRSUFAC_FILL
&Texture	IDM_BZRSUFAC_TEXTURE

按枚举变量中对应的项,响应上述诸消息:

```
void CCuvSufcView::OnBzrsufacMesh()
{
    // TODO: Add your command handler code here
    m_typeControl = BezierSurfaceMesh;
    Invalidate();
}

void CCuvSufcView::OnUpdateBzrsufacMesh(CCmdUI* pCmdUI)
{
    // TODO: Add your command update UI handler code here
    if(m_typeControl == BezierSurfaceMesh)
        pCmdUI->SetCheck(TRUE);
    else
        pCmdUI->SetCheck(FALSE);
}

void CCuvSufcView::OnBzrsufacFill()
{
    // TODO: Add your command handler code here
    m_typeControl = BezierSurfaceFill;
    Invalidate();
}

void CCuvSufcView::OnUpdateBzrsufacFill(CCmdUI* pCmdUI)
{
    // TODO: Add your command update UI handler code here
    if(m_typeControl == BezierSurfaceFill)
        pCmdUI->SetCheck(TRUE);
    else
        pCmdUI->SetCheck(FALSE);
```

```
}

void CCuvSufcView::OnBzrsufacTexture()
{
    // TODO: Add your command handler code here
    m_typeControl = BezierSurfaceTex;
    Invalidate();
}

void CCuvSufcView::OnUpdateBzrsufacTexture(CCmdUI* pCmdUI)
{
    // TODO: Add your command update UI handler code here
    if(m_typeControl == BezierSurfaceTex)
        pCmdUI->SetCheck(TRUE);
    else
        pCmdUI->SetCheck(FALSE);
}
```

(2) 添加保护成员函数

添加一个保护成员函数:BzrSufacMesh(),并编辑如下:

```
void CCuvSufcView::BzrSufacMesh()
{
    int i, j;
    //控制点
    GLfloat controlPoints[4][4][3] =
    {
      {{-0.8f, -0.6f, 0.8f},{-0.2f, -0.6f, 1.6f},{0.2f, -0.6f, -0.4f},{0.6f, -0.6f, 0.8f}},
      {{-0.6f, -0.2f, 0.8f},{-0.2f, -0.2f, 1.6f},{0.2f, -0.2f, -0.4f},{0.6f, -0.2f, 0.8f}},
      {{-0.6f, 0.2f, 0.8f}, {-0.2f, 0.2f, 0.4f}, {0.2f, 0.2f, 0.0f}, {0.3f, 0.2f, -0.4f}},
      {{-0.6f, 0.6f, 0.8f}, {-0.2f, 0.6f, 0.4f}, {-0.8f, 0.6f, 0.0f}, {0.8f, 0.6f, -0.4f}}
    };
    //二维评价器
    glMap2f(GL_MAP2_VERTEX_3, 0.0 1.0, 3 4, 0.0, 1.0, 12, 4,
            (const float *)controlPoints);
    //激活
    glEnable(GL_MAP2_VERTEX_3);
    glColor3f(0.0, 0.0, 1.0);
    //绘制
    glPushMatrix ();
        glRotatef(45.0, 1.0, 1.0, 1.0);
        for (j = 0; j <= 8; j++)
        {
            glBegin(GL_LINE_STRIP);
                for (i = 0; i <= 30; i++)
```

```
            glEvalCoord2f((GLfloat)i/30.0, (GLfloat)j/8.0);
        glEnd();
        glBegin(GL_LINE_STRIP);
            for (i = 0; i <= 30; i++)
            glEvalCoord2f((GLfloat)j/8.0, (GLfloat)i/30.0);
        glEnd();
    }
  glPopMatrix ();
  //挂起
  glDisable(GL_MAP2_VERTEX_3);
}
```

该函数将绘制一个网格曲面,因为 glBegin()函数的参数为 GL_LINE_STRIP。

(3) 编辑 DrawScene()函数

在语句 switch(m_typeControl){.....}中添加语句:

```
case    BezierSurfaceMesh:    BzrSufacMesh();    break;
```

(4) 运行程序

再次编译连接并运行程序,执行菜单 Surface \ Bezier \ Mesh,结果绘制了一个曲面,该曲面以网格形式绘制,如图 13-3。

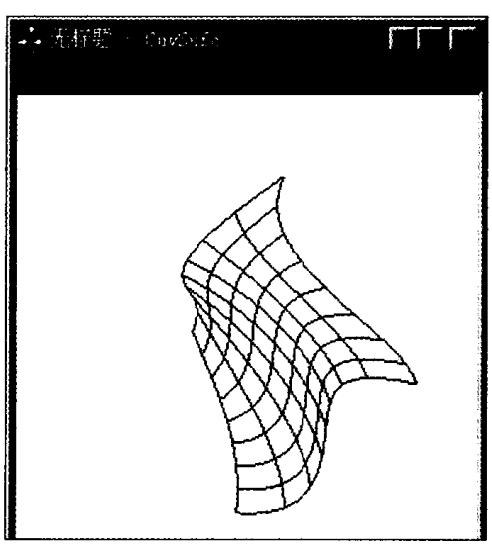

图 13-3 Bezier 网格曲面

13.4.3 绘制一个填充曲面

本节将绘制一个填充曲面。为了能显示曲面的空间感,还需要对其进行光照。同时本节将采用自动等分参数空间的方法,这需要 glMapGrid2f()和 glEvalMesh2f()函数配合使用。

(1) 增加两个保护成员函数

增加两个保护成员函数:Lighting()和 BzrSufacFill()函数。Lighting()函数处理光照,Bzr-

SufacFill()函数则绘制曲面。分别编辑如下：

```
void CCuvSufcView::Lighting()
{
    GLfloat lightAmb[] = { 0.2f, 0.2f, 0.2f, 1.0f };
    GLfloat lightPos[] = { 1.0f, 1.0f, 2.0f, 1.0f };
    GLfloat matDiffuse[] = { 0.0f, 0.7f, 0.7f, 1.0f };
    GLfloat matSpecular[] = { 1.0f, 1.0f, 1.0f, 1.0f };
    GLfloat matShininess[] = { 60.0f };
    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    glLightfv(GL_LIGHT0, GL_AMBIENT, lightAmb);
    glLightfv(GL_LIGHT0, GL_POSITION, lightPos);
    glMaterialfv(GL_FRONT, GL_DIFFUSE, matDiffuse);
    glMaterialfv(GL_FRONT, GL_SPECULAR, matSpecular);
    glMaterialfv(GL_FRONT, GL_SHININESS, matShininess);
}

void CCuvSufcView::BzrSufacFill()
{
    //控制点
    GLfloat controlPoints[4][4][3] =
    {
      {{-0.8f, -0.6f, 0.8f},{-0.2f, -0.6f, 1.6f},{0.2f, -0.6f, -0.4f},{0.6f, -0.6f, 0.8f}},
      {{-0.6f, -0.2f, 0.8f},{-0.2f, -0.2f, 1.6f},{0.2f, -0.2f, -0.4f},{0.6f, -0.2f, 0.8f}},
      {{-0.6f, 0.2f, 0.8f}, {-0.2f, 0.2f, 0.4f}, {0.2f, 0.2f, 0.0f}, {0.3f, 0.2f, -0.4f}},
      {{-0.6f, 0.6f, 0.8f}, {-0.2f, 0.6f, 0.4f}, {-0.8f, 0.6f, 0.0f}, {0.8f, 0.6f, -0.4f}}
    };
    //光照
    Lighting();
    glEnable(GL_AUTO_NORMAL);
    glEnable(GL_NORMALIZE);
    //二维评价器
    glMap2f(GL_MAP2_VERTEX_3, 0.0, 1.0, 3, 4,
            0.0, 1.0, 12, 4,
            (const float *)controlPoints);
    //自动网格
    glMapGrid2f(30, 0.0, 1.0, 30, 0.0, 1.0);
    //激活
    glEnable(GL_MAP2_VERTEX_3);
    //绘制
    glPushMatrix ();
        glRotatef(45.0, 1.0, 1.0, 1.0);
        glEvalMesh2(GL_FILL, 0, 30, 0, 30);
```

```
    glPopMatrix ();
    //挂起
    glDisable(GL_MAP2_VERTEX_3);
    glDisable(GL_AUTO_NORMAL);
    glDisable(GL_NORMALIZE);
    glDisable(GL_LIGHTING);
}
```

(3) 编辑 DrawScene()函数

在语句 switch(m_typeControl){.....}中添加语句:

```
case    BezierSurfaceFill:    BzrSufacFill();    break;
```

(4) 运行程序

再次编译连接并运行程序,执行菜单 Surface \ Bezier \ Fill,结果绘制了一个曲面,该曲面具有光感,如图 13-4。

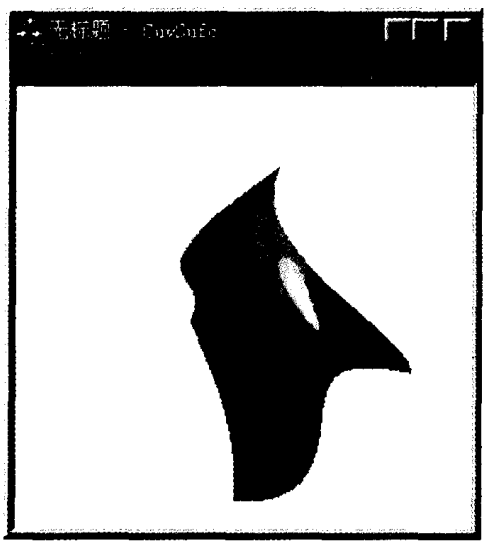

图 13-4 Bezier 填充曲面

与绘制 Bezier 曲线一样,在绘制 Bezier 曲面时,应首先给定控制曲面的控制点,然后将控制点信息输入到二维评价器,让其进行映射和计算,最后应用评价器所返回的曲面顶点的几何信息进行计算。

但是,用二维评价器进行曲面的绘制工作时,所需要处理的信息远远超过用它来绘制一维曲线所处理的信息。不仅要启动二维映射功能,还需要确定曲面的法线。确定一个 Bezier 曲面的法线方法大致有两种,其一是简单地执行如下语句:

```
glEnable(GL_AUTO_NORMAL);
glEnable(GL_NORMALIZE);
```

其二是给定控制 Bezier 曲面的法线控制点,将法线控制点作为二维评价器的输入信息,让其通过映射和计算为曲面产生法线:

```
glEnable(GL_MAP2_NORMAL);
glMap2f(GL_MAP2_NORMAL, 0.0, 1.0, 3, 4,
                        0.0, 1.0, 12, 4, (const float *)NormalControlPoints);
```

事实上,二维评价器具有十分强大的功能,我们还可以借助二维评价器为曲面产生纹理坐标:

```
glEnable(GL_MAP2_TEXTURE_COORD_3 );
glMap2f(GL_MAP2_TEXTURE_COORD_3 , 0.0, 1.0, 3, 4,
                        0.0, 1.0, 12, 4, (const float *)TextureControlPoints);
```

13.5 将位图贴到曲面上

本节的工作是将一个位图作为纹理,映射到曲面而生成纹理曲面。利用这种方法,可以制作草地等场景。将位图贴图到曲面时,另外要作的工作是:首先应生成纹理控制点序列,然后创建一个二维评价器,让其生成纹理坐标。将位图贴图映射到曲面的步骤是:

第一,生成几何控制点序列;

第二,生成纹理控制点序列;

第三,生成法线控制点序列(也可自动生成);

第四,进行纹理映射;

第五,分别创建二维评价器,生成几何顶点、纹理坐标和法线矢量(自动生成时,就不创建相应的评价器);

第六,进行光照;

第七,激活各种所需特定功能;

第八,绘制曲面;

第九,挂起各种已用特定功能。

事实上,制作纹理曲面是对纹理贴图和曲面生成的综合应用。下面,我们来完成本节的任务。

(1) 给工程添加外部文件

从 Texture 工程所在的目录拷贝 Dib.h、Dib.Cpp 和 studio.Bmp 等文件至 CuvSufc 工程所在的目录。为 CuvSufc 工程添加两个外部文件:Dib.h 和 Dib.Cpp。这相当于为该工程添加了一个处理位图的工具类:Cdib 类。

(2) 添加文件包含和公有成员变量

在文件 CuvSufcView.h 的头部添加文件包含:

```
#include      "Dib.h"
```

为 CuvSufcView 类添加公有成员变量:

```
CDib* m_pDib;
```

(3) 添加一个保护成员函数

添加一个保护成员函数:BzrSufacTex(),并编辑如下:

```
void CCuvSufcView::BzrSufacTex()
```

```
{
    //几何控制点
    GLfloat controlPoints[4][4][3] =
    {
      {{-0.8f, -0.6f, 0.8f},{-0.2f, -0.6f, 1.6f},{0.2f, -0.6f, -0.4f},{0.6f, -0.6f, 0.8f}},
      {{-0.6f, -0.2f, 0.8f},{-0.2f, -0.2f, 1.6f},{0.2f, -0.2f, -0.4f},{0.6f, -0.2f, 0.8f}},
      {{-0.6f, 0.2f, 0.8f}, {-0.2f, 0.2f, 0.4f}, {0.2f, 0.2f, 0.0f}, {0.3f, 0.2f, -0.4f}},
      {{-0.6f, 0.6f, 0.8f}, {-0.2f, 0.6f, 0.4f}, {-0.8f, 0.6f, 0.0f}, {0.8f, 0.6f, -0.4f}}
    };

    //纹理控制点
    GLfloat textureCntrlPnt[2][2][2] =
    {
        {{0.0f,0.0f}, {0.0f,0.1f}}, {{1.0f,0.0f}, {1.0f,1.0f}}
    };

    //构造纹理
    m_pDib = new CDib("studio.BMP");
    GLubyte * pTextureBits = (GLubyte * )m_pDib->GetDibBitsPtr();
    GLint width = m_pDib->GetDibWidth();
    GLint height = m_pDib->GetDibHeight();
    glPixelStorei(GL_UNPACK_ALIGNMENT,1);

    glTexImage2D(GL_TEXTURE_2D,0,3,width,height,
                 0,GL_BGR_EXT,GL_UNSIGNED_BYTE,pTextureBits);

    glTexParameterf(GL_TEXTURE_2D,GL_TEXTURE_WRAP_S,GL_CLAMP);
    glTexParameterf(GL_TEXTURE_2D,GL_TEXTURE_WRAP_T,GL_CLAMP);
    glTexParameterf(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,
                 GL_NEAREST);
    glTexParameterf(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,
                 GL_NEAREST);
    glTexEnvf(GL_TEXTURE_ENV,GL_TEXTURE_ENV_MODE,GL_DECAL);

    //二维评价器:    生成几何坐标和纹理坐标
    glMap2f(GL_MAP2_VERTEX_3, 0, 1, 3, 4,
            0, 1, 12, 4, (const float * )controlPoints);
    glMap2f(GL_MAP2_TEXTURE_COORD_2, 0, 1, 2, 2,
            0, 1, 4, 2, (const float * )textureCntrlPnt);
    //光照
    Lighting();

    glEnable(GL_TEXTURE_2D);
    glEnable(GL_MAP2_TEXTURE_COORD_2);
```

```
    glEnable(GL_MAP2_VERTEX_3);
    glEnable(GL_AUTO_NORMAL);
    glEnable(GL_NORMALIZE);

    //绘制
    glMapGrid2f(30, 0.0, 1.0, 30, 0.0, 1.0);
    glPushMatrix ();
        glScalef(1.2f,1.2f,1.2f);
        glRotatef(45.0f,0.0f,0.0f,1.0f);
        glRotatef(45.0f, 1.0f, 1.0f, 1.0f);
        glEvalMesh2(GL_FILL, 0, 30, 0, 30);
    glPopMatrix ();

    //挂起
    glDisable(GL_MAP2_VERTEX_3);
    glDisable(GL_MAP2_TEXTURE_COORD_2);
    glDisable(GL_TEXTURE_2D);
    glDisable(GL_AUTO_NORMAL);
    glDisable(GL_NORMALIZE);
    glDisable(GL_LIGHTING);
}
```

(4) 编辑 DrawScene()函数

在语句 switch(m_typeControl){.....}内添加语句：

```
case    BezierSurfaceTex:    BzrSufacTex();    break;
```

(5) 运行程序

再次编辑连接并运行程序，执行菜单 Surface \ Bezier \ Texture，最后图像被贴在曲面上，如图 13-5 所示。

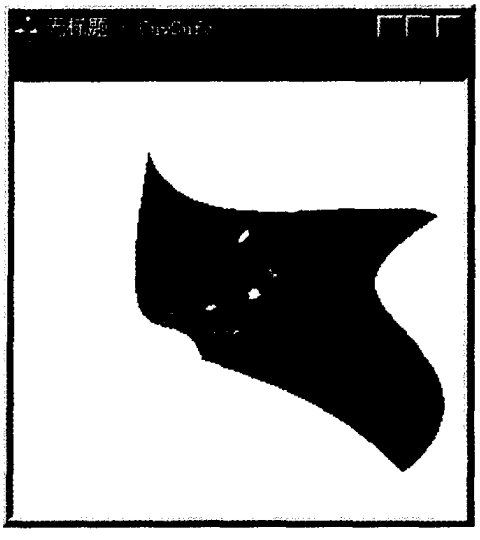

图 13-5　Bezier 纹理曲面

图像要正确地贴在曲面上一般是比较困难的。由于曲面不像平面那样可平展地放置一张图像,因而图像会发生拉伸、挤压和扭曲等变形。这些变形在一定程度上都是不可避免的,但是,可以调整纹理控制点序列来进行改善。

13.6 NURBS 曲面

本节将绘制 NURBS 曲面。NURBS 曲面的绘制与 NURBS 曲线的绘制过程是一致的。但是在绘制 NURBS 曲面时,由于 NURBS 曲面是二维参数曲面,因此需要在非均匀参数轴上定义两个节点控制序列。绘制一个 NURBS 曲面的步骤是:

第一,生成几何控制点序列及节点序列;

第二,生成纹理控制点序列及节点序列;

第三,生成法线控制点序列及节点序列(也可自动生成);

第四,创建 NURBS 对象并确定该对象的属性;

第五,进行纹理映射;

第六,进行光照;

第七,激活各种所需特定功能;

第八,绘制曲面;

第九,挂起各种已用特定功能。

绘制 NURBS 曲面的重要函数是:gluNurbsSurface(),其原型为:

```
void gluNurbsSurface( GLUnurbsObj * nobj, GLint sknot_count,
                      GLfloat * sknot, GLint tknot_count, GLfloat * tknot,
                      GLint s_stride, GLint t_stride, GLfloat * ctlarray,
                      GLint sorder, GLint torder, GLenum type );
```

其中,nobj 参数为一个 NURBS 对象;sknot_count 参数表示 u 参数方向的节点数目,sknot 参数则为其地址,对应序列是一个递增序列;tknot_count 参数表示 v 参数方向的节点数目,tknot 参数则为其地址,对应序列也是一个递增序列;s_stride,t_stride 参数分别表示 u 和 v 方向上数据偏移量;ctlarray 参数表示控制点序列,它可以表示几何控制点、纹理控制点或法线控制点等;sorder, torder 参数分别表示 u 和 v 参数轴 NURBS 曲线的阶;type 参数则表示与控制点序列相对应的输出数据类型。

与绘制 NURBS 曲线一样,gluNurbsSurface()函数必须在 gluBeginSurface() \ gluEndSurface()函数对中被调用。

下面我们来绘制一个 NURBS 曲面,并以网格、实填充和边界线方式来显示。

(1) 添加保护成员变量:

给 CCuvSufcView 类添加一个保护成员变量:

```
GLuint        m_nurbsSurfaceMode;
```

并初始化为:

```
m_nurbsSurfaceMode = GLU_OUTLINE_POLYGON;
```

该变量控制曲面的显示类型。

(2) 添加一个保护成员函数

添加一个保护成员函数:NurbsSurface(),并编辑如下:

```
void CCuvSufcView::NurbsSurface()
{
    //控制点
    GLfloat controlPoints[4][4][3] =
    {
      {{-0.8f, -0.6f, 0.8f},{-0.2f, -0.6f, 1.6f},{0.2f, -0.6f, -0.4f},{0.6f, -0.6f, 0.8f}},
      {{-0.6f, -0.2f, 0.8f},{-0.2f, -0.2f, 1.6f},{0.2f, -0.2f, -0.4f},{0.6f, -0.2f, 0.8f}},
      {{-0.6f, 0.2f, 0.8f}, {-0.2f, 0.2f, 0.4f}, {0.2, 0.2f, 0.0f}, {0.3f, 0.2f, -0.4f}},
      {{-0.6f, 0.6f, 0.8f}, {-0.2f, 0.6f, 0.4f}, {-0.8f, 0.6f, 0.0f}, {0.8f, 0.6f, -0.4f}}
    };
    //节点
    GLfloat sknots[8] = {0.0f, 0.0f, 0.0f, 0.0f, 1.0f, 1.0f, 1.0f, 1.0f};
    GLfloat tknots[8] = {0.0f, 0.0f, 0.0f, 0.0f, 1.0f, 1.0f, 1.0f, 1.0f};
    //光照
    Lighting();
    glDepthFunc(GL_LESS);
    glEnable(GL_AUTO_NORMAL);
    glEnable(GL_NORMALIZE);
    //NURBS 对象及属性
    GLUnurbsObj * theNurb;
    theNurb = gluNewNurbsRenderer();
    gluNurbsProperty (theNurb, GLU_SAMPLING_TOLERANCE, 25.0);
    gluNurbsProperty(theNurb, GLU_DISPLAY_MODE, m_nurbsSurfaceMode);
    //绘制
    glNewList (1, GL_COMPILE);
        gluBeginSurface(theNurb);
            gluNurbsSurface(theNurb,     8, sknots,     8, tknots,
                3,
                12,
                (float * )controlPoints,
                4, 4,
                GL_MAP2_VERTEX_3);
        gluEndSurface(theNurb);
    glEndList ();
    glPushMatrix();
        glScalef(1.2f,1.2f,1.2f);
        glRotatef(45.0f,0.0f,0.0f,1.0f);
        glRotatef(45.0f, 1.0f, 1.0f, 1.0f);
        glCallList (1);
    glPopMatrix();
    glDisable(GL_AUTO_NORMAL);
```

```
glDisable(GL_NORMALIZE);
glDisable(GL_LIGHTING);
```

该函数负责曲面的绘制,并由变量 m_nurbsSurfaceMode 控制显示方式。

(3) 添加菜单项

为 Surface 菜单添加一个菜单项,其名为 Nurbs,在其属性对话框的 Caption 栏中键入 Nu&rbs,并勾选 Pop-up 属性。现在为该菜单项添加 3 个子项,按表 13-8 完成。

表 13-8

Caption	ID
Mes&h	IDM_NURBSSUFAC_MESH
Fil&l	IDM_NURBSSUFAC_FILL
P&atch	IDM_NURBSSUFAC_PACH
Tri&m	IDM_NURBSSUFAC_TRIM

请响应上述诸消息:

```
void CCuvSufcView::OnNurbssufacMesh()
{
    m_typeControl = NurbsSurfaceMesh;
    m_nurbsSurfaceMode = GLU_OUTLINE_POLYGON;
    Invalidate();
}

void CCuvSufcView::OnUpdateNurbssufacMesh(CCmdUI* pCmdUI)
{
    // TODO: Add your command update UI handler code here
    if(m_typeControl == NurbsSurfaceMesh)
        pCmdUI->SetCheck(TRUE);
    else
        pCmdUI->SetCheck(FALSE);
}

void CCuvSufcView::OnNurbssufacFill()
{
    m_typeControl = NurbsSurfaceFill;
    m_nurbsSurfaceMode = GLU_FILL;
    Invalidate();
}

void CCuvSufcView::OnUpdateNurbssufacFill(CCmdUI* pCmdUI)
{
```

```
    // TODO: Add your command update UI handler code here
    if(m_typeControl == NurbsSurfaceFill)
        pCmdUI->SetCheck(TRUE);
    else
        pCmdUI->SetCheck(FALSE);
}

void CCuvSufcView::OnNurbssufacPach()
{
    m_typeControl = NurbsSurfacePach;
    m_nurbsSurfaceMode = GLU_OUTLINE_PATCH;
    Invalidate();
}

void CCuvSufcView::OnUpdateNurbssufacPach(CCmdUI* pCmdUI)
{
    // TODO: Add your command update UI handler code here
    if(m_typeControl == NurbsSurfacePach)
        pCmdUI->SetCheck(TRUE);
    else
        pCmdUI->SetCheck(FALSE);
}

void CCuvSufcView::OnNurbssufacTrim()
{
    m_typeControl = NurbsSurfaceTrim;
    Invalidate();
}

void CCuvSufcView::OnUpdateNurbssufacTrim(CCmdUI* pCmdUI)
{
    if(m_typeControl == NurbsSurfaceTrim)
        pCmdUI->SetCheck(TRUE);
    else
        pCmdUI->SetCheck(FALSE);
}
```

(4) 编辑 DrawScene()函数

在语句 switch(m_typeControl){...}中添加以下语句:

```
case NurbsSurfaceMesh:    NurbsSurface();    break;
case NurbsSurfaceFill:    NurbsSurface();    break;
case NurbsSurfacePach:    NurbsSurface();    break;
```

(5) 运行程序

再次编译连接并运行程序，执行菜单 Surface \ Nurbs \ Mesh，Surface \ Nurbs \ Fill 和 Surface \ Nurbs \ Patch。结果分别绘制一个网格曲面(如图 13-6)、实填充曲面(图略)和曲面边界线(图略)。

NurbsSurface()函数中，与用户相交互的语句是：

gluNurbsProperty(theNurb, GLU _ DISPLAY _ MODE, m _ nurbsSurfaceMode);

该语句通过成员变量 m _ nurbsSurfaceMode 选择 GLU _ OUTLINE _ POLYGON、GLU _ FILL 和 GLU _ OUTLINE _ PATCH 等 OpenGL 常量。这是一种交互的机制。OpenGL 提供了几百个常量，这些常量一般均可以通过控制变量来进行交互选择，或控制程序的执行，或控制图形的处理方式，或控制图形的输出形状等(参见第 2 章第 9 节)。

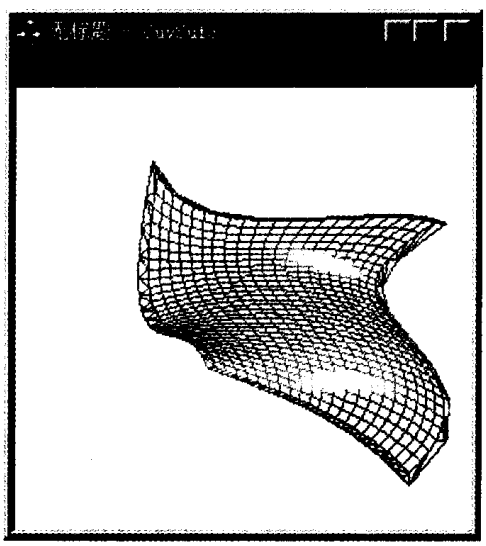

图 13-6　NURBS 网格曲面

最后指出，应用 NURBS 方法可以为曲面提供几何顶点颜色，为曲面提供法线和纹理坐标等。这些技术为高级制作提供了算法接口。

例如：

```
gluBeginSurface(theNurb);
    gluNurbsSurface(theNurb, 8, sknots,8, tknots,4,16,(float * )controlPoints _ Color,
                    4, 4, GL _ MAP2 _ COLOR _ 4);
    gluNurbsSurface(theNurb, 8, sknots,8, tknots,3,12,(float * )controlPoints,
                    4, 4, GL _ MAP2 _ VERTEX _ 3);
gluEndSurface(theNurb);
```

又如：

```
gluBeginSurface(theNurb);
    //gluNurbsSurface(theNurb, 8, sknots,8, tknots,4,16,(float * )controlPoints _ Color,
      //                4, 4, GL _ MAP2 _ COLOR _ 4);
    gluNurbsSurface(theNurb, 8, sknots,8, tknots,3,12,(float * )controlPoints _ Normal,
                    4, 4, GL _ MAP2 _ NORMAL );
    gluNurbsSurface(theNurb, 8, sknots,8, tknots,3,12,(float * )controlPoints _ Texture,
```

```
                4, 4, GL_MAP2_TEXTURE_COORD_3 );
    gluNurbsSurface(theNurb, 8, sknots,8, tknots,3,12,(float * )controlPoints,
                4, 4, GL_MAP2_VERTEX_3);
    gluEndSurface(theNurb);
```

13.7 曲面的剪切

OpenGL 提供了一个对曲面进行剪切的功能,利用这个功能,可以制作多连通曲面。多连通曲面是由外部边界线和内部边界线界定的曲面。当沿着曲面边界线行走时,曲面应在行人的左边,行人右边的洞或外部空间则不是多连通曲面的部分。这样,外部边界应是逆时针方向的,而内部边界则是顺时针方向的,更内层的边界方向可以类似确定。剪切区域定义了将要计算的 NURBS 表面域的子集,通过限制将要计算的区域,有可能生成包含洞的 NURBS 表面或生成有光滑边界的表面。剪切区域由一整套表面参数空间的封闭剪切循环定义。在对曲面进行剪切时,应先说明外边界剪切曲线,然后再说明内边界剪切曲线。剪切曲线可以由 gluPwlCurve()函数和 gluNurbsCurve()函数或其组合来定义,剪切曲线应是一个封闭曲线。

gluPwlCurve()函数描述一个分段线性 NURBS 曲线,其原型为:

```
void gluPwlCurve( GLUnurbsObj * nobj, GLint count, GLfloat * array,
                  GLint stride, GLenum type );
```

其中,nobj 参数为一个 NURBS 曲面对象;count 参数为剪切曲线上点的个数;array 参数表示剪切曲线控制点序列;stride 参数表示描述曲线点的分量个数;type 参数则表示参数空间曲线的维数,GLU_MAP1_TRIM_2 表示二维, GLU_MAP1_TRIM_3 表示三维。

gluPwlCurve()函数必须在 gluBeginTrim() \ gluEndTrim()函数对中被调用才有效。下面,我们来对本章的 NURBS 曲面进行剪切。

(1) 添加一个保护成员函数

为 CCuvSufcView 类添加一个保护成员函数:NurbsSufacTrim(),并编辑如下:

```
void CCuvSufcView::NurbsSufacTrim()
{
    //控制点
    GLfloat controlPoints[4][4][3] =
    {
      {{-0.8f, -0.6f, 0.8f},{-0.2f, -0.6f, 1.6f},{0.2f, -0.6f, -0.4f},{0.6f, -0.6f, 0.8f}},
      {{-0.6f, -0.2f, 0.8f},{-0.2f, -0.2f, 1.6f},{0.2f, -0.2f, -0.4f},{0.6f, -0.2f, 0.8f}},
      {{-0.6f,  0.2f, 0.8f},{-0.2f,  0.2f, 0.4f},{0.2f,  0.2f,  0.0f},{0.3f, 0.2f, -0.4f}},
      {{-0.6f,  0.6f, 0.8f},{-0.2f,  0.6f, 0.4f},{-0.8f, 0.6f,  0.0f},{0.8f, 0.6f, -0.4f}}
    };
    //节点
    GLfloat sknots[8] = {0.0f, 0.0f, 0.0f, 0.0f, 1.0f, 1.0f, 1.0f, 1.0f};
    GLfloat tknots[8] = {0.0f, 0.0f, 0.0f, 0.0f, 1.0f, 1.0f, 1.0f, 1.0f};
    //外边界:逆时针
    GLfloat edgePoint[5][2] =
    {
```

```
    {0.0f, 0.0f},    {1.0f, 0.0f},    {1.0f, 1.0f},    {0.0f, 1.0f},
};
//内边界:顺时针
GLfloat pwlPoint[5][2] =
{
    {0.75f, 0.75f},    {0.75f, 0.25f},  {0.25f, 0.25f},  {0.25f,0.75f},  {0.75f, 0.75f}
};
//光照
Lighting();

glDepthFunc(GL_LESS);
glEnable(GL_AUTO_NORMAL);
glEnable(GL_NORMALIZE);
//NURBS 对象及属性
GLUnurbsObj * theNurb;
theNurb = gluNewNurbsRenderer();
gluNurbsProperty (theNurb, GLU_SAMPLING_TOLERANCE, 25.0);
gluNurbsProperty(theNurb, GLU_DISPLAY_MODE, GLU_FILL);
//绘制
glNewList (1, GL_COMPILE);
    gluBeginSurface(theNurb);
        gluNurbsSurface(theNurb,
            8, sknots,
            8, tknots,
            3,
            12,
            (float * )controlPoints,
            4, 4,
            GL_MAP2_VERTEX_3);
        gluBeginTrim (theNurb);
            gluPwlCurve (theNurb, 5,
                (float * )edgePoint, 2,
                GLU_MAP1_TRIM_2);
        gluEndTrim (theNurb);
        gluBeginTrim (theNurb);
            gluPwlCurve (theNurb, 5,
                (float * )pwlPoint, 2,
                GLU_MAP1_TRIM_2);
        gluEndTrim (theNurb);
    gluEndSurface(theNurb);
glEndList ();

glPushMatrix();
    glScalef(1.2f,1.2f,1.2f);
```

```
        glRotatef(45.0f,0.0f,0.0f,1.0f);
        glRotatef(45.0f, 1.0f, 1.0f, 1.0f);
        glCallList (1);
    glPopMatrix();

    glDisable(GL_AUTO_NORMAL);
    glDisable(GL_NORMALIZE);
    glDisable(GL_LIGHTING);
}
```

(2) 编辑 DrawScene()函数

在 switch(m_typeControl)中添加以下语句:

```
case    NurbsSurfaceTrim:    NurbsSufacTrim();    break;
```

(3) 运行程序

再次编译连接并运行程序,执行菜单 Surface \ Nurbs \ Trim,结果对曲面进行了剪切,如图 13-7。

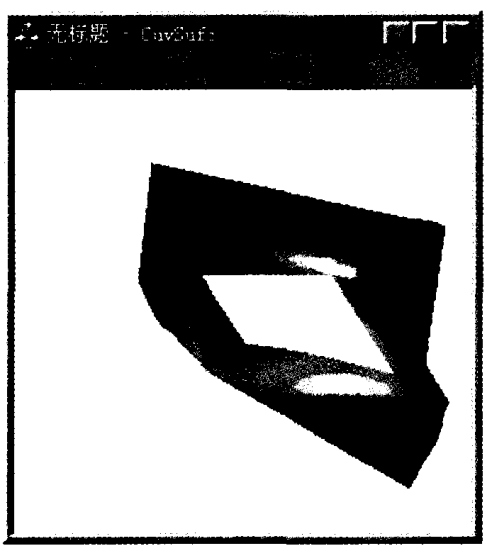

图 13-7 连通曲面

再一次强调,用于界定多连通曲面的曲线一定要是封闭的。edgePoint 和 pwlPoint 数组分别表示的 5 个点按顺序构成封闭的曲线。NURBS 曲线和分段线性 NURBS 曲线可以共同提供一个边界曲线,比如:

```
gluBeginTrim(nobj);
gluPwlCurve(. . . edgePoint_1, ,GLU_MAP1_TRIM_2);
gluNurbsCurve(. . . edgePoint_2, ,GLU_MAP1_TRIM_2);
gluNurbsCurve(. . . edgePoint_3,, GLU_MAP1_TRIM_3);
gluEndTrim(nobj);
```

但是应该注意,edgePoint_1 最后一个边界点应是 edgePoint_2 的第一个边界点,edgePoint_2 的最后一个边界点应是 edgePoint_3 的第一个边界点,依次类推。而且,曲线的类型

也可以不相同,例如,GLU _ MAP1 _ TRIM _ 2 和 GLU _ MAP1 _ TRIM _ 3 常量可以分别同时使用。用分段线性 NURBS 曲线和用 NURBS 曲线所界定的多连通曲面最后生成的结果有可能不同。

第 14 章　高级制作技术

前面章节已经介绍了如何处理 OpenGL 的几何图元和像素数据。这两类数据最终都经过光栅化以及基片操作生成对应屏幕像素的各种信息，这些信息包括像素的颜色、深度以及其他辅助信息。像素的每种信息都加以分类，用位平面的形式存储起来。存储这些信息的区域称为缓存。OpenGL 的缓存包括颜色缓存、深度缓存、模板缓存和累积缓存。第 7 章已经介绍了如何操作颜色缓存和深度缓存，本章主要讲述模板缓存和累积缓存的应用。

14.1　模板缓存及其应用

14.1.1　模板缓存的操作

模板缓存存放像素的模板值。模板值可用于控制像素是否被改写。因而模板缓存可以禁止在屏幕的某些区域绘图。可以像操作颜色缓存和深度缓存一样操作模板缓存。

(1) 清除模板缓存

可用下述语句来完成：

```
glClear(GL_STENCIL_BUFFER_BIT);
```

(2) 设置清除值

可以用 glClearStencil()函数设置模板缓存的清除值，比如：

```
glClearStencil(1);
```

则清除模板缓存的第一个位平面的值。

(3) 屏蔽模板缓存

用 glStencilMask()函数控制模板缓存位面的单个位的写属性。缺省值均为 1，即各位均可写。

(4) 模板测试

模板测试将一个像素的模板参考值和即将写入的同一像素的模板值进行比较，如果满足条件，则该像素的模板值被代替并允许参与下一级测试，否则将不再显示。模板测试的激活和挂起可用如下语句实现：

```
glEnable(GL_STENCIL_TEST);
glDisable(GL_STENCIL_TEST);
```

glStencilFunc()函数确定模板测试的条件，其原型为：

```
void glStencilFunc( GLenum func, GLint ref, GLuint mask );
```

其中，func 参数表示测试条件，可以为 GL_NEVER、GL_LESS、GL_LEQUAL 、GL_GREATER、GL_GEQUAL 、GL_EQUAL 、GL_NOTEQUAL 及 GL_ALWAYS。ref 参数为

模板测试的参考值,取值范围为[0,2^n-1],n 为模板缓存的位平面数。mask 参数表示当最终检测完成之后,它与存储的模板值和参考值进行逻辑与运算,最后再按测试条件参与比较。只有通过了模板测试,像素才能进入光栅化操作的下一步骤。

(5) 设置模板测试行为

不论模板测试是否通过,可用 glStencilOp()函数来控制模板测试行为和模板值。该函数的原型为:

void glStencilOp(GLenum fail, GLenum zfail, GLenum zpass);

其中, fail 参数表示模板测试失败时,应该怎样改变像素的模板值。它可以取表 14-1 中的值。

表 14-1

常 量	含 义
GL_KEEP	保留当前值
GL_ZERO	模板值被设置为零
GL_REPLACE	设为 glStencilFunc 定义的参考值
GL_INCR	当前值加 1
GL_DECR	当前值减 1
GL_INVERT	当前值按位取反

zfail 参数表示模板测试通过但深度测试失败时,如何改变像素的模板值,取值与 fail 相同。zpass 参数表示当模板测试和深度测试均获得通过,或者当模板测试通过但无深度缓存或不允许进行深度测试时,应该怎样改变像素的模板值,取值也与 fail 相同。

14.1.2 模板缓存的应用

模板缓存给我们提供了一个非常灵活且实用的绘制特殊图形的方法。模板就相当于一块掩膜,可以在掩膜上绘制,也可以禁止在掩膜上输出图形。重要的是掩膜可以自己定义,可以是一块规则的平面区域,也可以是一块不规则的平面区域或空间区域。一块掩膜具有相同的模板值,这样整个用户区就被分割成多个区域,可以在不同的区域绘制指定的图形。因而模板缓存具有以下重要的用途:

第一,屏蔽屏幕区域(凸区域或凹区域,因而也可以绘制凹多边形等);

第二,遮盖物体;

第三,制作物体的交集。

本节将制作两段代码,第一段代码首先生成一块正方形掩膜,将一个球画在这块掩膜上,并绘制另一个球的不在这块掩膜上的部分。第二段代码说明怎样将两个物体的交集制作成为掩膜,然后在其上绘制其他物体。我们将在编程的过程中详细讲述其用法。下面我们来完成编程工作。

(1) 极小化工程

建立一个名为 Buffer 的工程,并按第 5 章中的方法极小化 OpenGL。

(2) 添加枚举变量及保护成员变量

在 Buffer View.h 文件的头部添加:

```
enum{StencilMask, StencilClip,    AntiAccum,   NonAntiAccum,   Focus};
```

增加一个保护成员变量：

```
GLint m_typeControl;
```

初始化为：

```
m_typeControl = StencilMask;
```

该变量控制显示类别。

(3) 添加保护成员函数

添加两个保护成员函数，StencilAppMask()和 StencilAppClip()函数。StencilAppMask()函数创建一块正方形掩膜，将球绘制其上，StencilAppClip()函数制作两球的交集，并用此交集作为掩膜对一个相当大的四边形进行高级剪切。分别编辑如下：

```
void CBufferView::StencilAppMask()
{
    GLfloat matDiffuse[] = { 0.8f, 0.8f, 0.0f, 1.0f };
    GLfloat matSpecular[] = { 1.0f, 1.0f, 1.0f, 1.0f };
    GLfloat maskDiffuse[] = { 0.1f, 0.2f, 0.8f, 1.0f };
    GLfloat maskSpecular[] = { 0.1, 1.0, 1.0, 1.0f };
    GLfloat lightPosition[] = { 0.0f, 3.0f, 2.0f, 0.0f };
    glLightfv(GL_LIGHT0, GL_POSITION, lightPosition);
    GLint iMatList = glGenLists(2);

    glNewList(iMatList, GL_COMPILE);
        glMaterialfv(GL_FRONT, GL_DIFFUSE, maskDiffuse);
        glMaterialfv(GL_FRONT, GL_SPECULAR,maskSpecular);
        glMaterialf(GL_FRONT, GL_SHININESS, 25.0f);
    glEndList();
    glNewList(iMatList + 1, GL_COMPILE);
        glMaterialfv(GL_FRONT, GL_DIFFUSE, matDiffuse);
        glMaterialfv(GL_FRONT, GL_SPECULAR, matSpecular);
        glMaterialf(GL_FRONT, GL_SHININESS, 70.0f);
    glEndList();
    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    glDepthFunc(GL_LESS);
    //创建一个正方形模板区域
    glStencilFunc (GL_ALWAYS, 0x1, 0x1);
    glStencilOp (GL_REPLACE, GL_REPLACE, GL_REPLACE);
    glBegin(GL_QUADS);
        glVertex3f (-0.7f, 0.0f, 0.0f);
        glVertex3f (0.0f, 0.7f, 0.0f);
        glVertex3f (0.7f, 0.0f, 0.0f);
```

```
        glVertex3f (0.0f, -0.7f, 0.0f);
    glEnd();
    //    在模板值为 1 的地方绘制一个蓝色球
    glStencilFunc (GL_EQUAL, 0x1, 0x1);
    glCallList (iMatList);
    DrawSphere(0.5);
    //    在模板值非 1 的地方绘制一个黄色球
    glStencilFunc (GL_NOTEQUAL, 0x1, 0x1);
    glStencilOp (GL_KEEP, GL_KEEP, GL_KEEP);
    glPushMatrix();
        glCallList (iMatList + 1);
        DrawSphere(0.9);
    glPopMatrix();
}

void CBufferView::StencilAppClip()
{
    //    第一句
    glDepthFunc(GL_LEQUAL);
    glColorMask(GL_FALSE,GL_FALSE,GL_FALSE,GL_FALSE);
    glStencilFunc (GL_ALWAYS, 0x1, 0x3);
    glStencilOp (GL_REPLACE, GL_REPLACE, GL_REPLACE);
    glPushMatrix();
        glTranslatef(-0.4f,0.0f,0.0f);
        DrawSphere(0.8);
    glPopMatrix();
    glStencilFunc (GL_EQUAL, 0x1, 0x3);
    glStencilOp (GL_KEEP, GL_INCR, GL_INCR);
    //    第十一句
    glPushMatrix();
        glTranslatef(0.4f,0.0f,0.0f);
        DrawSphere(0.8);
    glPopMatrix();
    glColorMask(GL_TRUE,GL_TRUE,GL_TRUE,GL_TRUE);
    glStencilFunc (GL_EQUAL, 0x2, 0x3);
    glStencilOp (GL_KEEP, GL_KEEP, GL_KEEP);
    glPolygonMode(GL_FRONT,GL_FILL);
    glDisable(GL_LIGHTING);
    //    第二十句
    glPushMatrix();
        glTranslatef(0.0f,0.0f,1.0f);
        glBegin(GL_QUADS);
            glColor3f(1.0f,1.0f,1.0f);
            glVertex3f (-0.8f, 0.8f, 0.0f);
```

```
            glVertex3f (0.8f, 0.8f, 0.0f);
            glColor3f(0.0f,0.0f,0.0f);
            glVertex3f (0.8f, -0.8f, 0.0f);
            glVertex3f (-0.8f, -0.8f, 0.0f);
        glEnd();
    glPopMatrix();
}
```

下面,我们对 StencilAppClip()函数进行详细分析。第一句,表示只有深度值满足"小于或等于"的条件才获得通过。此句给出深度测试的条件,适用于 StencilAppClip()函数的所有深度测试。第二句,将颜色缓存屏蔽以禁止向颜色缓存写入颜色信息,这样可以集中计算资源来处理模板缓存,并给不同的屏幕区域赋模板值。第三句,设置模板测试总能够获得通过,并且只比较最低的两位(3 的二进制为 11)。注意,在对 PIXELFORMATDESCRIPTOR 结构进行初始化时,并没有设置模板缓存的位数。第四句,将获得通过的像素模板值设为参考值,即为 1。这时,程序将执行第五至第八句,进行第一个球的绘制工作。由于模板测试总能通过,因而凡是满足深度测试条件的球的部分所对应的屏幕像素空间的模板值均被置为 1。但是,此时颜色缓存是禁止写入的,因而事实上是借球的绘制确定了一个模板,这个模板是模板值为 1 的像素的集合。第九句,重新设置模板测试条件,即凡是模板值等于 1 的像素才获准通过。第十句,如果通过模板测试和深度测试,则对应像素的模板值就会自动加 1,即等于 2。按第九和第十句设定的条件和操作行为,由第十一至第十四句进行第二个球的绘制。很显然,位于第二个球上对应像素的模板值为 1 的点必然为两球相交的点。这样两球相交的区域对应的像素的模板值就为 2。第十五句,则解除对颜色缓存写的禁令以便能在屏幕的任何区域绘制图形。由于模板测试已经启动,因而绘图就会受到限制,也正是由于有这种限制,才给我们的高级制作带来了工具。第十六句,表明只能在模板值为 2 的区域绘制图形。第十七句,指示获得通过测试的像素的模板值不会改变。第二十句至第三十句,则对一个较大的正方形进行剪切,只留下像素的模板值为 2 的区域。该区域看起来似两球之交。其中,第二十一句是比较重要的,因为模板值为 2 的区域并非一个平面区域,该句就使正方形通过深度测试。将会看到,由 DrawScene()函数的语句:glClearStencil(0x0);设置其他区域的模板值为 0。

(4) 添加菜单

添加一个名为 Stencil 的菜单,即在属性对话框的 Caption 栏内加入 S&tencil。增加以下菜单项,表 14-2 是它们的属性对话框中 Caption 及 ID 号。

表 14-2

Caption	ID
&Mask	IDM_STENCIL_MASK
C&lip	IDM_STENCIL_CLIP

用常用的方法响应上述诸消息:

```
void CBufferView::OnStencilMask()
{
    m_typeControl = StencilMask;
```

```
    Invalidate();
}

void CBufferView::OnUpdateStencilMask(CCmdUI* pCmdUI)
{
    if(m_typeControl == StencilMask)
        pCmdUI->SetCheck(TRUE);
    else
        pCmdUI->SetCheck(FALSE);
}

void CBufferView::OnStencilClip()
{
    m_typeControl = StencilClip;
    Invalidate();
}

void CBufferView::OnUpdateStencilClip(CCmdUI* pCmdUI)
{
    if(m_typeControl == StencilClip)
        pCmdUI->SetCheck(TRUE);
    else
        pCmdUI->SetCheck(FALSE);
}
```

(5) 编辑 DrawScene()函数

```
void CBufferView::DrawScene()
{
    glClearColor(1.0f,1.0f,1.0f,1.0f);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glClearStencil(0x0);    //用模板值 0 来清除模板缓存
    glClear(GL_STENCIL_BUFFER_BIT);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glTranslatef(0.0f,0.0f,-4.5f);
    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    glEnable(GL_STENCIL_TEST);
    switch(m_typeControl)
    {
        case StencilMask:    StencilAppMask();    break;
        case StencilClip:    StencilAppClip();    break;
    }
```

```
    glDisable(GL_STENCIL_TEST);
    glDisable(GL_LIGHTING);
    glFinish();
}
```

(6) 运行程序

编译连接并运行程序，执行菜单 Stencil \ Mask 和 Stencil \ Clip。要注意只有重绘用户区时，才能在模板缓存中绘制图形，因而第一次执行菜单 Stencil \ Mask 和以后执行该菜单的结果略有不同，分别如图 14-1 和图 14-2 所示。

必须指出，OpenGL 还提供了一个快速的剪切函数 glScissor()。该函数在窗口内确定一个矩形区域，只有在该区域内的像素才有可能获得通过。该函数的缺点是被剪切的区域只能是矩形。注意，事实上这是为像素执行剪切测试，为此，应先用 glEnable()函数及 GL_SCISSOR_TEST 参数允许剪切测试。

请把如下语句放在 OnDraw()函数的适当位置，观察运行的结果(注：检测之后，请将它们一并删除)：

```
glEnable(GL_SCISSOR_TEST);
glScissor(-0.3,0.3,0.6,-0.6);
glDisable(GL_SCISSOR_TEST);
```

利用颜色缓存、深度缓存和模板缓存的各种操作与组合可以解决许多技术问题，其综合应用需要在编程的过程中逐步发现、积累、分析、总结和升华。

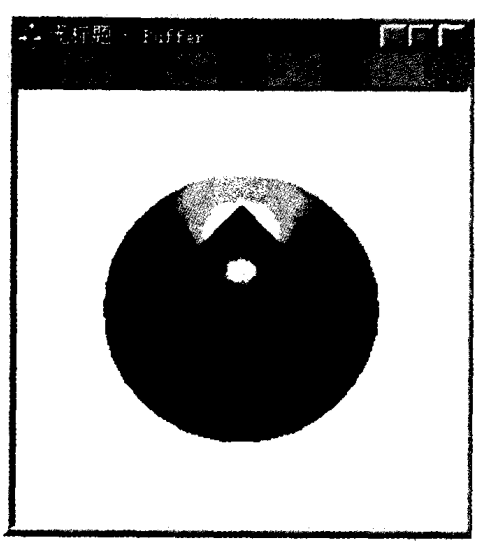

图 14-1 掩膜

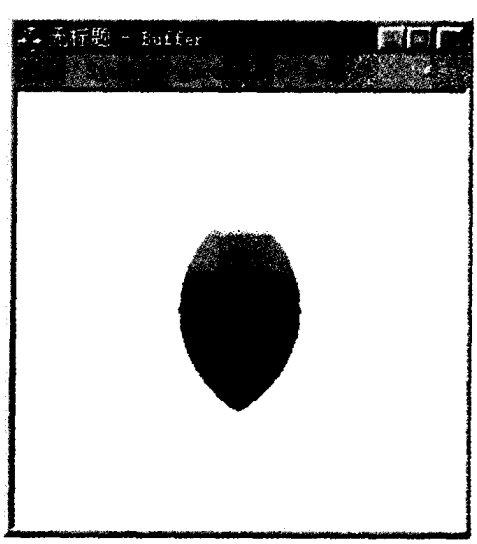

图 14-2 高级剪切

14.2 累积缓存及其应用

累积缓存利用一系列的绘制结果来实现场景的反走样、景深模拟、运动模糊等。借助于累积缓存，多次绘制场景，每次绘制时都轻微地移动场景(相当于在空间上抖动场景)，把多次绘

制的结果进行累积并最后一次输出,结果场景的边界会变得模糊,从而实现全局反走样。景深模拟就是对照相技术的模拟。一般来说,离焦点越近,物体就越清晰,离焦点越远,景深长,物体就越模糊。OpenGL 并不直接对整个场景进行对焦处理,这样绘制的物体不论远近均有相同的清晰度。雾化作为一种特殊的营造环境氛围的手段,雾化了整个场景,但并不能达到对焦的效果。OpenGL 可以利用累积缓存来解决这个问题。为了获得景深效果,将视景体作一些扰动(相当于在空间上抖动场景),并把这些扰动累积于累积缓存,最后一次写入颜色缓存,获得对焦效果。对象的运动模糊是用于经过屏幕的运动对象并使其模糊的特殊效果。运动模糊常用于物体的移动比换帧速度快的运动情况,在动画设计中,常用运动模糊让对象产生比实际运动更快的感觉,或者通过运动的模糊产生特殊的外观。OpenGL 利用累积缓存将物体的当前位置和前一帧的位置进行模糊计算(相当于在时间上抖动部分物体),然后把计算结果从累积缓存传输到颜色缓存来实现运动模糊的特殊效果。本书通过例程只展示全局反走样技术和景深模拟技术。

14.2.1 累积缓存的操作

下面我们介绍有关累积缓存的操作。

(1) 清除累积缓存:

```
glClear(GL_ACCUM_BUFFER_BIT);
```

(2) 设置清除颜色

用 glClearAccum()函数来设置累积缓存的清除颜色,比如:

```
glClearAccum(1.0f,0.0f,0.0f,1.0f);
```

(3) 累积行为

用 glAccum()函数控制对累积缓存的操作,其原型为:

```
void glAccum( GLenum op,GLfloat value );
```

其中,op 参数为操作行为,可以为 GL_ACCUM、GL_LOAD、GL_ADD 、GL_MULT 或 GL_RETURN 。下面分别介绍其含义。

GL_ACCUM:从当前选中的颜色缓存中读取像素,像素的每个颜色分量分别乘以 value,并将计算结果累加到累积缓存中去。

GL_LOAD:用计算结果替代累积缓存中的内容。

GL_ADD:将累积缓存中像素的颜色分量分别加上 value,结果放入累积缓存中。

GL_MULT:从累积缓存中读取像素的颜色值,将颜色分量乘以 value,结果放入累积缓存中。

GL_RETURN:从累积缓存中读取像素,将颜色分量乘以 value,结果放入颜色缓存,实现累积结果的绘制。

value 参数则只是一个操作因子。

14.2.2 全局反走样

本节介绍怎样利用累积缓存进行全局反走样。首先用显示列表来管理场景。场景由 5 个

颜色不同的球组成,每个球将以刻面的形式绘出,最后将全局反走样结果和没有进行全局反走样的结果进行对比。

(1) 添加一个宏定义

在 BufferView.h 文件的开头部分定义:

```
# define     ACCUMLIST          1
```

(2) 添加 3 个保护成员函数

为 CBufferView 类添加 3 个保护成员函数:RenderObjects(),AccumlateScene()和 NonAccumuScene()函数。RenderObjects()函数负责场景的绘制:

```
void CBufferView::RenderObjects()
{
    GLfloat matDif[4] = {0.6f, 0.05f, 0.05f, 1.0f};
    GLfloat matSpe[4] = {1.0f, 1.0f, 1.0f, 1.0f};
    glNewList(2,GL_COMPILE);
        glMaterialfv(GL_FRONT, GL_DIFFUSE, matDif);
        glMaterialfv(GL_FRONT, GL_SPECULAR, matSpe);
        glMaterialf(GL_FRONT, GL_SHININESS, 75.0f);
    glEndList();
    matDif[0] = 0.75f;          matDif[1] = 0.7f;          matDif[2] = 0.2f;
    glNewList(3,GL_COMPILE);
        glMaterialfv(GL_FRONT, GL_DIFFUSE, matDif);
        glMaterialfv(GL_FRONT, GL_SPECULAR, matSpe);
        glMaterialf(GL_FRONT, GL_SHININESS, 60.0f);
    glEndList();
    matDif[0] = 0.65f;          matDif[1] = 0.65f;          matDif[2] = 0.65f;
    glNewList(4,GL_COMPILE);
        glMaterialfv(GL_FRONT, GL_DIFFUSE, matDif);
        glMaterialfv(GL_FRONT, GL_SPECULAR, matSpe);
        glMaterialf(GL_FRONT, GL_SHININESS, 60.0f);
    glEndList();
    matDif[0] = 0.08f;          matDif[1] = 0.75f;          matDif[2] = 0.08f;
    glNewList(5,GL_COMPILE);
        glMaterialfv(GL_FRONT, GL_DIFFUSE, matDif);
        glMaterialfv(GL_FRONT, GL_SPECULAR, matSpe);
        glMaterialf(GL_FRONT, GL_SHININESS, 30.0f);
    glEndList();
    matDif[0] = 0.05f;          matDif[1] = 0.65f;          matDif[2] = 0.65f;
    glNewList(6,GL_COMPILE);
        glMaterialfv(GL_FRONT, GL_DIFFUSE, matDif);
        glMaterialfv(GL_FRONT, GL_SPECULAR, matSpe);
        glMaterialf(GL_FRONT, GL_SHININESS, 50.0f);
    glEndList();
    GLUquadricObj * quadObj;
```

```
        quadObj = gluNewQuadric();
        gluQuadricDrawStyle(quadObj,GLU_FILL);
        gluQuadricNormals(quadObj,GLU_SMOOTH);
        glNewList((ACCUMLIST),GL_COMPILE);
            glPushMatrix();
                glTranslatef(-0.8f,-0.3f,1.0f);
                glCallList(2);
                gluSphere(quadObj,0.4,16,16);
                glTranslatef(0.6f,0.0f,-0.7f);
                glCallList(3);
                gluSphere(quadObj,0.4,16,16);
                glTranslatef(0.6f,0.0f,-0.7f);
                glCallList(4);
                gluSphere(quadObj,0.4,16,16);
                glTranslatef(0.6f,0.0f,-0.7f);
                glCallList(5);
                gluSphere(quadObj,0.4,16,16);
                glTranslatef(0.6f,0.0f,-0.7f);
                glCallList(6);
                gluSphere(quadObj,0.4,16,16);
            glPopMatrix();
        glEndList();
}
```

AccumlateScene()函数进行反走样处理：

```
void CBufferView::AccumlateScene()
{
        GLdouble dOffsetX[16], dOffsetY[16];
        //    定义场景偏移量
        GLdouble dTransRef[16][2] =
        {
          {0.4375, 0.4375},{0.1875, 0.5625},{0.9375, 1.1875},{0.4375, -0.065},
          {0.6875, 0.5625},{0.1875, 0.0625},{0.6875, 0.3125},{0.1875, 0.3125},
          {0.4375, 0.1875},{-0.065, 0.4375}{0.6875, 0.8125},{0.4375, 0.6875},
          {0.6875, 0.0625},{0.9375, 0.9375},{1.1875, 0.8125},
         };
        GLint i,     viewport[4];
        GLfloat lightPosition[] = { 0.0f, 3.0f, 2.0f, 0.0f };
        glLightfv(GL_LIGHT0, GL_POSITION, lightPosition);
        //    获取视窗大小
        glGetIntegerv (GL_VIEWPORT, viewport);
        //    计算场景真实偏移量
        for (i = 0; i < 16; i++)
        {
```

```
        dOffsetX[i] = dTransRef[i][0] * 10/viewport[2];
        dOffsetY[i] = dTransRef[i][1] * 10/viewport[3];
    }
    glShadeModel(GL_FLAT);
    //清除累积缓存
    glClearAccum(0.0, 0.0, 0.0, 0.0);
    glClear(GL_ACCUM_BUFFER_BIT);
    //    将场景重新绘制 16 次,并累积在累积缓存中
    for (i = 0; i < 16; i++)
    {
        glPushMatrix();
            glTranslated(dOffsetX[i], dOffsetY[i], 0.0);
            glClear(GL_DEPTH_BUFFER_BIT | GL_COLOR_BUFFER_BIT);
            glCallList((ACCUMLIST));
        glPopMatrix();
        glAccum(GL_ACCUM, 1.0/16.0);
        glFlush();
        SwapBuffers(wglGetCurrentDC());
    }
    //    一次性返回累积缓存的结果
    glAccum(GL_RETURN, 1.0);
    glShadeModel(GL_SMOOTH);
}
```

下面,我们对这一函数进行分析,最重要的是下述语句的设计问题:

```
for (i = 0; i < 16; i++)
{
    glPushMatrix();
        glTranslated(dOffsetX[i], dOffsetY[i], 0.0);
        glClear(GL_DEPTH_BUFFER_BIT | GL_COLOR_BUFFER_BIT);
        glCallList((ACCUMLIST));
    glPopMatrix();
    glAccum(GL_ACCUM, 1.0/16.0);
    glFlush();
    SwapBuffers(wglGetCurrentDC());
}
```

glTranslated()函数将场景进行平移,其平移量由 dOffsetX 和 dOffsetY 数组提供。平移量一般可以按一定的算法来生成,比如,噪化算法。当然也可以按经验来提供。glClear()函数对深度和颜色缓存进行清除处理,这是十分必要的。如果不这样,由于深度缓存和颜色缓存都没有进行屏蔽(事实上,由于是重新绘制整个场景,也不能进行屏蔽),每次将进行累加,结果将会不正确。场景的重新绘制工作则由语句 glCallList((ACCUMLIST));来完成。由于进行了变换,用矩阵堆栈来进行管理也是必要的。glAccum(GL_ACCUM, 1.0/16.0);语句则累积 1/16 的结果入颜色缓存。当累积完成之后,应由 glFlush()函数迫使 OpenGL 重新绘制一遍场

景，并通过 SwapBuffers()来交换前后台缓存，这样我们可以清楚地看到场景的重新绘制过程。

NonAccumuScene()函数则不进行全局反走样，以进行比较分析：

```
void CBufferView::NonAccumuScene()
{
    glShadeModel(GL_FLAT);
    glCallList((ACCUMLIST));
    glShadeModel(GL_SMOOTH);
}
```

(3) 优化显示列表

本书我们已经多次用到显示列表，也学习了显示列表的多种用法。在已编辑过的程序中真正发挥作用的还较少，这主要是由于显示列表大都仅被调用一次，有的地方刚建立就马上调用。在 AccumlateScene()函数中，调用显示列表就达 16 次之多。因此，有必要只进行一次编译这个显示列表就可永远调用它，最好的办法就是将其放入 WM_CREAT 消息响应函数，因为该函数在整个程序运行过程中只执行一次，而且先于 OnDraw()函数执行。

编辑 OnCreate()函数，在语句 Init(); 之后添加：

```
RenderObjects();
```

最后，该函数为：

```
int CBufferView::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    if (CView::OnCreate(lpCreateStruct) == -1)
        return -1;
    Init();
    RenderObjects();
    return 0;
}
```

(4) 添加菜单

添加一个名为 Accumulate 的菜单，即在属性对话框的 Caption 栏内加入 &Accumulate。增加以下菜单项，表 14-3 是它们的属性对话框中的 Caption 及 ID 号。

表 14-3

Caption	ID
&Antialias	IDM_ANTISCENE
&Nonantialias	IDM_NON_ANTIACCUM
F&ocus Depth	IDM_FOCUS_DEPTH

用常用的方法响应上述诸消息：

```
void CBufferView::OnAntiscene()
{
    m_typeControl = AntiAccum;
```

```
    Invalidate();
}

void CBufferView::OnUpdateAntiscene(CCmdUI* pCmdUI)
{
    if(m_typeControl == AntiAccum)
        pCmdUI->SetCheck(TRUE);
    else
        pCmdUI->SetCheck(FALSE);
}

void CBufferView::OnNonAntiaccum()
{
    m_typeControl = NonAntiAccum;
    Invalidate();
}

void CBufferView::OnUpdateNonAntiaccum(CCmdUI* pCmdUI)
{
    if(m_typeControl == NonAntiAccum)
        pCmdUI->SetCheck(TRUE);
    else
        pCmdUI->SetCheck(FALSE);
}

void CBufferView::OnFocusDepth()
{
    m_typeControl = Focus;
    Invalidate();
}

void CBufferView::OnUpdateFocusDepth(CCmdUI* pCmdUI)
{
    if(m_typeControl == Focus)
        pCmdUI->SetCheck(TRUE);
    else
        pCmdUI->SetCheck(FALSE);
}
```

(5) 编辑 DrawScene()函数

在语句 switch(m_typeControl){... ...}中添加：

```
case AntiAccum:      AccumlateScene();      break;
case NonAntiAccum:   NonAccumuScene();      break;
```

(6) 运行程序

再次编译连接并运行程序,执行菜单 Accumulate \ Antialias 和 Accumulate \ Nonantialias,如图 14-3 和图 14-4 所示,并请比较两者之间的区别。

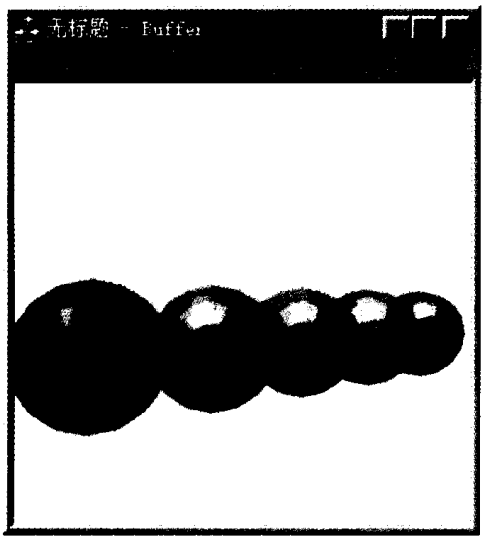

图 14-3 全局反走样场景

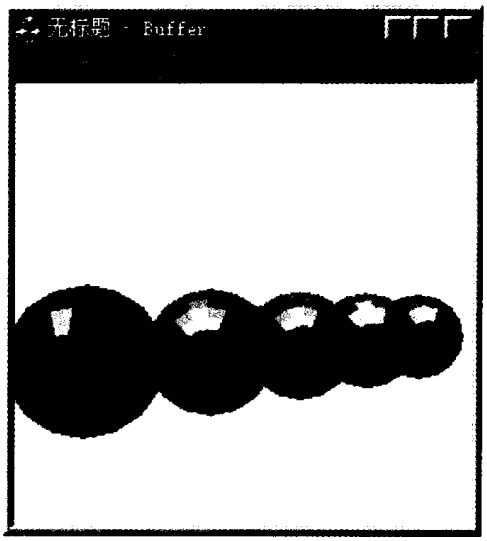

图 14-4 非反走样场景

14.2.3 景深模拟

景深模拟就是模拟聚焦的过程。在 OpenGL 中,对场景进行多次取景,每次取景时保持视景体的大小不改变,但视景体可进行前后上下左右轻微地移动,每取一次景就绘制一次场景,并把绘制结果放入累积缓存,最后一次输出累积结果。现在,我们对已绘制过的由 5 个球组成的场景进行景深模拟,请按以下步骤完成:

(1) 添加保护成员函数

添加 3 个保护成员函数:myFrustum()、myPerspective()和 FocusDepth()。myFrustum()函数设置 OpenGL 的 glFrustum()函数的参数,myPerspective()函数自定义视景体,并把视景体的参数传递给 myFrustum()函数。FocusDepth()函数则绘制整个场景并实现景深模拟。现分别编辑如下:

```
void CBufferView::myFrustum(GLdouble left,
        GLdouble right,    GLdouble bottom,
        GLdouble top,      GLdouble znear,
        GLdouble zfar,     GLdouble eyedx,
        GLdouble eyedy,    GLdouble focus )
{
    GLdouble xwsize, ywsize;
    GLdouble dx, dy;
    xwsize = right - left;
    ywsize = top - bottom;
```

```
    dx = -eyedx * znear/focus;
    dy = -eyedy * znear/focus;

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glFrustum(left + dx, right + dx, bottom + dy, top + dy, znear, zfar);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glTranslatef (-eyedx, -eyedy, 0.0);
}
```

在该函数中由 glFrustum()函数和 glTranslatef()函数将视景体和场景移动。由于每次要调整视景体,因而在此必须启动透视和模型视变换,并对相应的矩阵进行单位初始化。

myPerspective()函数的编辑如下:

```
void CBufferView::myPerspective(GLdouble fovy,
        GLdouble aspect,    GLdouble znear,
        GLdouble zfar,      GLdouble eyedx,
        GLdouble eyedy,     GLdouble focus )
{
    GLdouble fov,left,right,bottom,top;
    fov = ((fovy * 3.1415926535898) / 180.0) / 2.0;
    top = znear / (cos(fov) / sin(fov));
    bottom = -top;
    right = top * aspect;
    left = -right;
    myFrustum (left, right, bottom, top, znear, zfar,eyedx, eyedy, focus);
}
```

FocusDepth()函数的编辑如下:

```
void CBufferView::FocusDepth()
{
    typedef struct {
        GLfloat x, y;
    } OFFSET;

    OFFSET      eyeOffset[] =
    {
        {-0.33, 0.44},    {0.28, -0.39},
        {0.46, 0.14},     {-0.41, -0.19},
        {-0.18, 0.08},    {-0.08, -0.32},
        {0.10, 0.30},     {0.16, -0.05}
    };
    int iTimes;
    GLint viewport[4];
```

```
    glGetIntegerv (GL_VIEWPORT, viewport);
    if (! viewport[3]) return;
    glShadeModel(GL_FLAT);
    glClearAccum(0.0, 0.0, 0.0, 0.0);
    glClear(GL_ACCUM_BUFFER_BIT);
    for (iTimes = 0; iTimes < 8; iTimes++)
    {
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
        myPerspective (38.0,
              (GLdouble) viewport[2]/(GLdouble) viewport[3], 3, 7.0,
              0.04 * eyeOffset[iTimes].x, 0.04 * eyeOffset[iTimes].y, 2.7);
        glPushMatrix();
              glTranslatef(0.0f,0.0f, -4.5f);
              glCallList((ACCUMLIST));
        glPopMatrix();
        glAccum (GL_ACCUM, 0.125);
        glFlush();
        SwapBuffers(wglGetCurrentDC());
    }
    glAccum (GL_RETURN, 1.0);
    glFlush();
    glShadeModel(GL_SMOOTH);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glFrustum(-1.0,1.0,-1.0,1.0,3.0,7.0);
}
```

很显然,这里的处理方法与 AccumlateScene()函数的处理方法不一样,它不是简单地将场景平移,但可以认为是将照相机多次抖动并聚焦的结果。在重新绘制场景前,每次由 myPerspective()函数将视景体进行抖动,完成此工作时,应该还没有将照相机向视景方向推进,因而语句 glTranslatef(0.0f,0.0f,-4.5f);是十分必要的,由该语句将场景推入进视景体。然后,程序完成重新绘制场景的工作。由于每次均重新架设过相机,也就是说,每次都重新进行了透视变换,因而,必须重新启动透视变换:

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glFrustum(-1.0,1.0,-1.0,1.0,3.0,7.0);
```

景深模拟技术事实上是对焦技术思想的一种应用。

在函数 FocusDepth()中,下述语句

```
GLint viewport[4];
glGetIntegerv (GL_VIEWPORT, viewport);
```

可获得窗口的大小。

(2) 编辑 DrawScene()函数

在 switch(m _ typeControl){... ...}中添加如下语句:

```
case Focus:          FocusDepth();          break;
```

(3) 添加头文件

在文件的开头部分添加头文件:

```
# include        "math.h"
```

(4) 运行程序

再次编译连接并运行程序,执行菜单 Accumulate \ Focus Depth,则实现了对景深的模拟,如图 14-5 所示。请自行将其与全局反走样结果做比较。

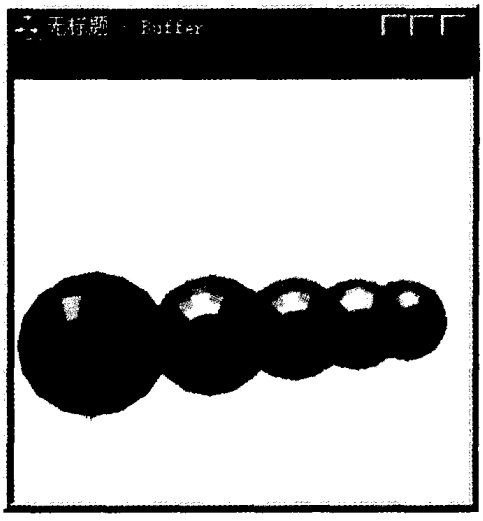

图 14-5 景深模拟

第 15 章　交互技术

应用 OpenGL 不仅可以控制物体的绘制，还可以对物体进行标记以区分在指定区域上绘制了哪些物体以及返回绘图信息。绘制物体是在 OpenGL 缺省的绘图模式下进行的，而标记物体和反馈信息则分别在选择模式和反馈模式下完成。绘图模式、选择模式和反馈模式构成了 OpenGL 开发三维图形应用程序的基础。在绘图模式中，物体的几何参数、光照与材质参数、纹理参数等等构成了交互式程序的绘制基础，选择模式则为用户提供了一种拾取物体的机制，因而成为交互式程序设计的重要方法。反馈模式则将绘图信息加以组织并返回给应用程序，成为程序设计的重要资料和用户需知的信息。

15.1　物体的选择

选择物体事实上是响应 OpenGL 应用程序的一个拾取事件，该事件通常是由鼠标来触发的。为此，应用程序必须建立物体的名称，并将名称加以组织。然后当发生拾取事件时，就对其进行响应。当拾取一个物体后，就以记录的形式组织相关信息并返回给应用程序。

15.1.1　建立名称堆栈

正如我们用矩阵堆栈来管理物体在空间的相互位置和层次关系一样，名称堆栈可以对物体进行命名，并通过名称堆栈建立物体的名称层次关系以最终标记物体的各个组成部分。

在建立名称堆栈前，首先应初始化名称堆栈，然后将名称压入名称堆栈和弹出名称堆栈。在操作名称堆栈时，还可以用指定的名称来替换堆栈顶层中的名称。这些操作可由以下 4 个函数来完成：glInitNames()，glPushName()，glPopName()及 glLoadName()。例如：

```
glInitNames();
glPushName((unsigned) - 1);
glLoadName (0);
glPushName(0);
    DrawObject_01();
glPopName();
glPushName(1);
    DrawObject_02();
glPopName();
glLoadName (1);
glPushName(0);
    DrawObject_11();
glPopName();
glPushName(1);
    DrawObject_12();
```

glPopName();

glPushName()和 glLoadName ()函数以一个无符号整数为参数,这个整数就是物体的名称。可以用 glPushName()/glPopName()函数对来建立多层次的名称堆栈。

15.1.2 拾取物体

拾取物体最重要的是要确定鼠标所在的位置是哪一个物体,这就需要定义一个有效的拾取区域。比如,鼠标热点周围的 5 个像素构成的区域为有效区时,则在此区域外的物体就不会获得被拾取的机会。

在 OpenGL 中,拾取物体是利用拾取矩阵和投影变换,将绘图和拾取范围限制在鼠标热点的有效区域中,一旦触发鼠标事件就进入选择模式并将其初始化,最后利用拾取矩阵拾取有效区的物体。有效拾取区的定义由 gluPickMatrix()函数来完成,其原型为:

```
void gluPickMatrix( GLdouble x, GLdouble y, GLdouble width, GLdouble height,
                    GLint viewport[4] );
```

其中,x,y 参数为拾取区域的中心,通常就是鼠标热点的窗口坐标;width,height 参数为拾取区域的宽度和高度;viewport 参数是存储窗口大小信息的地址。

一旦拾取成功,就以记录的形式返回与拾取物体相关的信息。生成一个记录表示一个物体被命中。一个命中记录由以下几项组成:首先是名称堆栈中名称的个数;紧接着是被命中的物体位于视景体内的部分的几何顶点的最大和最小窗口坐标的深度值;然后就是名称堆栈中名称的内容,其顺序是按进入名称堆栈的顺序排列的。

必须指出的是,尽管选择模式给程序提供了一个重要的交互手段,但要正确地命中物体却有许多问题需要解决:比如,当有效拾取区有许多物体时怎样从中选择一个或多个物体;当多个物体具有同一名称时又怎样处理物体;如果在有效区中不能命中物体又怎样来解决等等。这些问题均属于技术问题,可以基于 OpenGL 的选择模式,也可以构造自己的图形学算法来解决它们。

15.1.3 选择模式的应用

下面我们通过一个简单的例子来介绍应用选择模式的基本方法。这个例子是将用户区分成 4 个部分,当鼠标位于其中某一部分时,只要单击鼠标左按钮,就会以不同的颜色重新绘制这个区域的背景颜色。

应用选择模式的基本步骤可概括如下:

第一,创建存储返回命中记录的数组,由 glSelectBuffer ()函数来实现;

第二,进入选择模式,由 glRenderMode ()函数来决定;

第三,初始化名称堆栈;

第四,定义用于选择模式的视景体,拾取区域;

第五,建立名称堆栈;

第六,退出选择模式。

选择模式是由外部事件来触发的,当退出时就可以决定是否有物体被命中。由于选择模

式不向帧缓存写入绘图信息,在应用选择模式前必须要绘制一次场景。请按如下步骤来完成编程工作:

(1) 建立一个名为 SlctFdbk 的工程,并按第 5 章中的方法极小化 OpenGL。

(2) 添加枚举变量及保护成员变量

在 SlctFdbkView.h 文件的头部添加:

```
enum{Select,    Feedback};
```

增加保护成员变量:

```
GLint BackColor[2][2],          m_modeControl;
```

初始化为:

```
for (int i = 0; i < 2; i++)
for (int j = 0; j < 2; j ++)
    BackColor[i][j] = 0;

m_modeControl= Select;
```

m_modeControl 变量控制模式类别。BackColor 数组则存储 4 个背景区域的颜色。

(3) 添加保护成员函数

添加 5 个保护成员函数:CreateSpheres()、Circle()、DrawNonSelect()、DrawNameObjects()和 WhatHits()。CreateSpheres()函数创建光照球,DrawNameObjects()函数负责绘制物体和命名物体,Circle()函数创建轨道,DrawNonSelect()函数则显示轨道,而 WhatHits()函数则处理命中记录,分别编辑如下:

```
void CSlctFdbkView::CreateSpheres()
{
    GLfloat matDif_1[] = {0.8f, 0.8f, 0.0f, 1.0f};
    GLfloat matSpe_1[] = {1.0f, 1.0f, 1.0f, 1.0f};
    GLfloat matDif_2[] = {0.1f, 0.2f, 0.8f, 1.0f};
    GLfloat matSpe_2[] = {0.1, 1.0, 1.0, 1.0f};
    GLfloat matDif_3[] = {0.1f, 0.8f, 0.8f, 1.0f};
    GLfloat matSpe_3[] = {0.1, 1.0, 1.0, 1.0f};
    GLfloat matDif_4[] = {0.8f, 0.2f, 0.8f, 1.0f};
    GLfloat matSpe_4[] = {0.1, 1.0, 1.0, 1.0f};
    GLfloat lightPosition[] = {0.0f, 3.0f, 2.0f, 0.0f};

    glLightfv(GL_LIGHT0, GL_POSITION, lightPosition);

    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);

    glNewList(5, GL_COMPILE);
        glMaterialfv(GL_FRONT, GL_DIFFUSE, matDif_1);
```

```
        glMaterialfv(GL_FRONT, GL_SPECULAR,matSpe_1);
        glMaterialf(GL_FRONT, GL_SHININESS, 25.0f);
    glEndList();

    glNewList(6, GL_COMPILE);
        glMaterialfv(GL_FRONT, GL_DIFFUSE, matDif_2);
        glMaterialfv(GL_FRONT, GL_SPECULAR, matSpe_2);
        glMaterialf(GL_FRONT, GL_SHININESS, 70.0f);
    glEndList();

    glNewList(7, GL_COMPILE);
        glMaterialfv(GL_FRONT, GL_DIFFUSE, matDif_3);
        glMaterialfv(GL_FRONT, GL_SPECULAR, matSpe_3);
        glMaterialf(GL_FRONT, GL_SHININESS, 50.0f);
    glEndList();

    glNewList(8, GL_COMPILE);
        glMaterialfv(GL_FRONT, GL_DIFFUSE, matDif_4);
        glMaterialfv(GL_FRONT, GL_SPECULAR, matSpe_4);
        glMaterialf(GL_FRONT, GL_SHININESS, 60.0f);
    glEndList();

    GLUquadricObj * quadObj;
    glNewList(1,GL_COMPILE);
            glCallList(5);
            quadObj = gluNewQuadric();
            gluQuadricDrawStyle(quadObj,GLU_FILL);
            gluQuadricNormals(quadObj,GLU_SMOOTH);
            gluSphere(quadObj,0.2,16,16);
    glEndList();

    glNewList(2,GL_COMPILE);
            glCallList(6);
            quadObj = gluNewQuadric();
            gluQuadricDrawStyle(quadObj,GLU_FILL);
            gluQuadricNormals(quadObj,GLU_SMOOTH);
            gluSphere(quadObj,0.2,16,16);
    glEndList();

    glNewList(3,GL_COMPILE);
            glCallList(7);
            quadObj = gluNewQuadric();
            gluQuadricDrawStyle(quadObj,GLU_FILL);
            gluQuadricNormals(quadObj,GLU_SMOOTH);
```

```
            gluSphere(quadObj,0.2,16,16);
glEndList();

glNewList(4,GL_COMPILE);
            glCallList(8);
            quadObj = gluNewQuadric();
            gluQuadricDrawStyle(quadObj,GLU_FILL);
            gluQuadricNormals(quadObj,GLU_SMOOTH);
            gluSphere(quadObj,0.2,16,16);
glEndList();

glNewList(9,GL_COMPILE);
    glPushMatrix();
            glTranslatef(-0.4f,0.4f,0.0f);
            glCallList(1);
            glTranslatef(-0.6f,0.6f,0.0f);
            glCallList(4);
    glPopMatrix();
glEndList();

glNewList(10,GL_COMPILE);
    glPushMatrix();
            glTranslatef(0.4f,0.4f,0.0f);
            glCallList(2);
            glTranslatef(0.6f,0.6f,0.0f);
            glCallList(3);
    glPopMatrix();
glEndList();

glNewList(11,GL_COMPILE);
    glPushMatrix();
            glTranslatef(-0.4f,-0.4f,0.0f);
            glCallList(3);
            glTranslatef(-0.6f,-0.6f,0.0f);
            glCallList(2);
    glPopMatrix();
glEndList();

glNewList(12,GL_COMPILE);
    glPushMatrix();
            glTranslatef(0.4f,-0.4f,0.0f);
            glCallList(4);
            glTranslatef(0.6f,-0.6f,0.0f);
            glCallList(1);
```

```
        glPopMatrix();
    glEndList();

    glDisable(GL_LIGHTING);
}

void CSlctFdbkView::Circle(GLfloat dRadius, GLfloat fLine, GLfloat n)
{
    GLfloat dStep = (2 * 3.14159265)/(GLfloat)n;
    GLfloat fV[3];

    glLineWidth(fLine);

    glBegin(GL_LINE_STRIP);

    for(GLdouble i = 0.0;i <= 2 * 3.14159265 + dStep;i += dStep)
    {
        fV[0] = dRadius * cos(i);
        fV[1] = dRadius * sin(i);
        fV[2] = 0.0f;
        glVertex3fv(fV);
    }
    glEnd();
}

void CSlctFdbkView::DrawNameObjects(GLenum mode)
{
    int i,j;
    for(i = 0;i < 2;i++)
    {
        if (mode == GL_SELECT)glLoadName (i);
        for(j = 0;j < 2;j++)
        {
            if (mode == GL_SELECT)glPushName (j);

            glColor3f ((GLfloat) i/3.0, (GLfloat) j/3.0,
                (GLfloat) BackColor[i][j]/3.0);

            glRectf ((i-1) * 1.5f, (j-1) * 1.5f, i * 1.5f, j * 1.5f);

            if((i == 0)&&(j == 0))
            {
                glEnable(GL_LIGHTING);
                glCallList(9);
```

```
            glDisable(GL_LIGHTING);
        }

        if((i==0)&&(j==1))
        {
            glEnable(GL_LIGHTING);
            glCallList(10);
            glDisable(GL_LIGHTING);
        }

        if((i==1)&&(j==0))
        {
            glEnable(GL_LIGHTING);
            glCallList(11);
            glDisable(GL_LIGHTING);
        }

        if((i==1)&&(j==1))
        {
            glEnable(GL_LIGHTING);
            glCallList(12);
            glDisable(GL_LIGHTING);
        }
    }

    if (mode == GL_SELECT)glPopName();
    if (mode == GL_SELECT)glPopName();
  }
}
```

该函数将屏幕分成4部分。在每一部分,绘制一个填充四边形及两个球。它们共用两个名字,因而将名称分成了两组,且每组的深度均为2。

```
void CSlctFdbkView::WhatHits(GLint hits, GLuint * buffer)
{
    int i, j;
    GLuint ii, jj, names, *ptr;

    ptr = buffer;

    for (i = 0; i < hits; i++)
    {
        names = *ptr;
        ptr++;      ptr++;      ptr++;
```

```
        for (j = 0; j < names; j++)
        {
            if (j == 0) ii = *ptr;
            else if (j == names-1) jj = *ptr;

            ptr++;
        }
        BackColor[ii][jj] = (BackColor[ii][jj] + 1) % 4;
    }
}
```

WhatHits()函数处理返回记录。其中,hits 参数为命中记录的个数,buffer 参数为命中记录的数据地址。变量 names 为名称堆栈中物体名称的个数。在第一个 for 循环内部,由三条 ptr++;语句将指针指向名称堆栈的位置,并由第二个 for 循环来逐个读取名称堆栈中的内容,堆栈层次越深,names 值就越大。当 j=0 时,*ptr 事实上就是 glLoadName (i)中的 i 值。

该程序最后由 BackColor 数组来存储改变的颜色,这样能够判断是否已经作出了响应。

```
void CSlctFdbkView::DrawNonSelect()
{
    GLfloat lw[2];
    glGetFloatv(GL_LINE_WIDTH_RANGE,lw);

    glPushMatrix();
    glTranslatef(0.0f,0.0f,0.01f);
        glColor3f(1.0f,0.0f,0.0f);
        Circle(0.2,lw[1]/8.0,50);
        Circle(0.4,lw[1]/8.0,50);
        Circle(0.6,lw[1]/8.0,50);
        Circle(0.8,lw[1]/8.0,100);
        glColor3f(0.0f,1.0f,0.0f);
        Circle(1.0,lw[1]/8.0,100);
        Circle(1.2,lw[1]/8.0,100);
        Circle(1.4,lw[1]/8.0,100);
        Circle(1.6,lw[1]/8.0,100);
        glColor3f(0.0f,0.0f,1.0f);
        Circle(1.8,lw[1]/8.0,100);
        Circle(2.0,lw[1]/8.0,100);
        Circle(2.2,lw[1]/8.0,100);
        Circle(2.4,lw[1]/8.0,100);
        Circle(2.6,lw[1]/8.0,100);
        Circle(2.8,lw[1]/8.0,100);
    glPopMatrix();
}
```

(4) 增加 Windows 消息

添加一个响应鼠标左按钮被按下的消息 WM _ LBUTTONDOWN,并对该消息进行响应,删除 Visual C++ 已为我们生成的诸语句,重新编辑如下:

```
void CSlctFdbkView::OnLButtonDown(UINT nFlags, CPoint point)
{
//    第一条语句:
      if(m_modeControl == Feedback)return;

      GLuint selectBuf[512];
      GLint hits;
      GLint viewport[4];
      int x, y;

      x = point.x;
      y = point.y;
      glGetIntegerv (GL_VIEWPORT, viewport);

//    第九条语句:
      glSelectBuffer (512, selectBuf);
      glRenderMode (GL_SELECT);

      glInitNames();
      glPushName((unsigned) -1);

      glMatrixMode (GL_PROJECTION);
      glPushMatrix ();
          glLoadIdentity ();
          gluPickMatrix ((GLdouble) x, (GLdouble) (viewport[3] - y),
                5.0, 5.0, viewport);
          glFrustum(-1.0,1.0,-1.0,1.0,2.0,8.0);

          glMatrixMode(GL_MODELVIEW);
          glLoadIdentity();
          glTranslatef(0.0f,0.0f,-3.5f);
          DrawNameObjects (GL_SELECT);
          glMatrixMode (GL_PROJECTION);
      glPopMatrix ();

//    二十四条语句:
      hits = glRenderMode (GL_RENDER);
      WhatHits (hits, selectBuf);
      Invalidate();
}
```

其中,变量 selectBuf 用来存储命中记录,x, y 获取鼠标的位置,hits 为命中记录的个数,

viewport 存储视区的位置和大小等信息。在程序中,第九条语句指定存储命中记录的数组,并由第十条语句进入选择模式。在进入选择模式前一定要先指定存储命中记录的数组。第十一条语句初始化名称堆栈,这时名称堆栈是空的,由第十二条语句压入一个名称,其值为 -1,这样名称堆栈非空,也避免了以后可能出现的错误。第十三至第十七条语句定义有效拾取区域和用于选择的特定视景体。第十八至第二十一条语句观察物体,命名物体。第二十四条语句重新进入绘图模式,这时 glRenderMode()函数用 GL_RENDER 作为参数,返回的却是命中记录的个数,因为在进入绘图模式前 OpenGL 处于选择模式。第二十五条语句处理返回的命中记录的信息。

(5) 添加菜单

添加一个名为 Mode 的菜单,即在属性对话框的 Caption 栏内加入 &Mode。增加以下菜单项,表 15-1 是它们的属性对话框中的 Caption 及 ID 号。

表 15-1

Caption	ID
S&elect	IDM_SELECT
Fee&dback	IDM_FEEDBACK

用常用的方法响应上述诸消息:

```
void CSlctFdbkView::OnSelect()
{
    m_modeControl = Select;
    Invalidate();
}

void CSlctFdbkView::OnUpdateSelect(CCmdUI* pCmdUI)
{
    if(m_modeControl == Select)
        pCmdUI->SetCheck(TRUE);
    else
        pCmdUI->SetCheck(FALSE);
}

void CSlctFdbkView::OnFeedback()
{
    m_modeControl = Feedback;
    Invalidate();
}

void CSlctFdbkView::OnUpdateFeedback(CCmdUI* pCmdUI)
{
    if(m_modeControl == Feedback)
```

```
        pCmdUI->SetCheck(TRUE);
    else
        pCmdUI->SetCheck(FALSE);
}
```

(6) 编辑 DrawScene()函数

```
void CSlctFdbkView::DrawScene()
{
    glClearColor(1.0f,1.0f,1.0f,1.0f);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glTranslatef(0.0f,0.0f,-4.5f);

    glDepthFunc(GL_LESS);
    glDepthRange (0.0, 1.0);

    switch(m_modeControl)
    {
        case Select:        DrawNameObjects (GL_RENDER); break;
    }

    if(m_modeControl==Select)DrawNonSelect();
    glFinish();
}
```

(7) 编辑 OnCreate()函数

```
int CSlctFdbkView::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    if (CView::OnCreate(lpCreateStruct) == -1)
        return -1;

    Init();
    CreateSpheres();
    return 0;
}
```

利用该函数优化显示列表。

(8) 增加文件包含

在文件的头部添加:

```
#include "math.h"
```

(9) 运行程序

编译连接并运行程序，执行菜单 Mode \ Select。如图 15-1 所示。可见，视区被分成 4 个区域，每个区域标有不同的颜色，单击鼠标左按钮，相应区域会改变颜色。

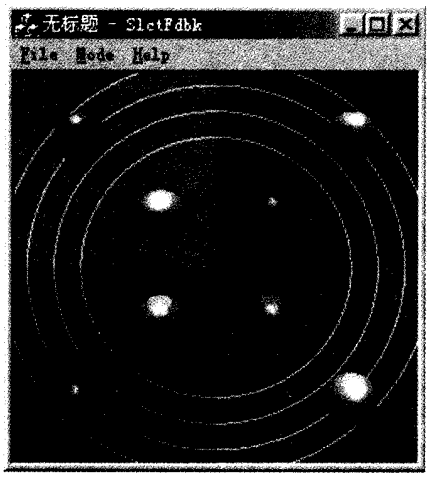

图 15-1　选择模式

15.2　信息反馈

15.2.1　反馈模式

信息反馈为程序设计提供了重要的运行资料。在反馈模式下，每个被光栅化的基元均产生一组数据，并由 OpenGL 把它们存放入反馈数组中。每组数据均有一个标记，该标记说明了基元的类型，接着是描述基元的顶点坐标值、其他如颜色和纹理等相关数据。在退出反馈模式之后，这些数据就被写入到反馈数组中。在应用反馈模式时，还可以插入一个标记，通过标记可以把一组数据与另一组数据区分开来，以便于识别和处理。

与选择模式相同的是，反馈模式不对像素进行任何光栅化操作，也不向帧缓存写入任何绘图信息，因而在应用反馈模式之前，必须绘制一次需要反馈信息的物体。与选择模式不同的是，反馈模式是根据基元被处理的情况决定返回已经被处理的几何顶点的相关数据，应用时要比选择模式简单得多，也不存在较大的技术问题。遗憾的是，OpenGL 直接返回的却是顶点的已经被光栅化后的窗口坐标(当然，可通过一系列线性变换获得该顶点的世界坐标)。反馈模式不像选择模式那样需要外设来触发，可以通过函数调用来实现。

应用反馈模式的基本步骤如下：

第一，确定反馈信息的信息类别和反馈数组。反馈数组应是一个存储浮点值的数组，由 glFeedbackBuffer()函数来完成，其原型为：

```
void glFeedbackBuffer( GLsizei size, GLenum type, GLfloat * buffer );
```

其中，size 参数表示能够写进反馈数组的数据的最大个数。type 参数为一个标记常数，它表明 OpenGL 应该返回顶点的哪些数据，可以取的值如下：GL _ 2D，GL _ 3D，GL _ 3D _ COLOR，

GL_3D_COLOR_TEXTURE 及 GL_4D_COLOR_TEXTURE。其含义说明如下：

GL_2D：只返回顶点的几何坐标(x,y)。

GL_3D：只返回顶点的几何坐标(x,y,z)。

GL_3D_COLOR：返回顶点的几何坐标(x,y,z)及顶点的颜色。若颜色为 RGBA 模式，则颜色分量的个数为 4，这样该组数据的个数为 7。

GL_3D_COLOR_TEXTURE：返回顶点的几何坐标(x,y,z)、顶点的颜色和纹理值。纹理由 4 个量组成。这样在 RGBA 模式下，该组数据的个数为 11 个。

GL_4D_COLOR_TEXTURE：返回顶点的被剪切后的几何坐标(x,y,z,w)、顶点的颜色和纹理值。这样在 RGBA 模式下，该组数据的个数为 12 个。

第二，将 glRenderMode ()函数的参数设为 GL_FEEDBACK，使 OpenGL 进入反馈模式。

第三，重新绘制物体，并按要求插入标记。重新绘制物体时，并不向帧缓存写入绘图信息。在反馈数组中，每组数据均有一个标记，这个标记说明了这组反馈信息描述的是基本几何对象，还是像素数据。这个标记可以是表 15-2 中的一些常数。

表 15-2

标　记	处理的基本对象
GL_POINT_TOKEN	顶点
GL_LINE_TOKEN	线
GL_LINE_RESET_TOKEN	线
GL_POLYGON_TOKEN	多边形
GL_BITMAP_TOKEN	位图
GL_DRAW_PIXEL_TOKEN	像素
GL_COPY_PIXEL_TOKEN	像素
GL_PASS_THROUGH_TOKEN	通过

反馈模式下，每次调用 glDrawPixels()或 glCopyPixels()函数都会向反馈数组中写入相关数据。但是，插入标记却是另一回事，它在一组数据与另一组数据间插入一个标记数组。由 glPassThrough ()函数来完成，并由其所在位置决定插在何处。比如：glPassThrough (2.0)，则生成一个标记组，该组由两项组成，第一项也为一个标记，即

GL_PASS_THROUGH_TOKEN，第二项为一个浮点数，此时即为 2.0。

第四，退出反馈模式，处理反馈数据。

15.2.2　反馈模式的应用

现在以一个具体的例子说明应用反馈模式的基本方法。该程序在场景中仅绘制 3 个点，给这 3 个点加上光照但没有纹理信息。在第一和第 2 个点之间插入一个标记，第二和第三个点间也插入一个标记。在绘制时，让第 2 个点位于视景体之外，尽管它也处于反馈模式中却不会反馈它的相关信息。请按如下步骤完成：

(1) 添加保护成员变量

为 CSlctFdbkView 类添加一个保护成员变量：

```
GLfloat        y;
```

在构造函数中初始化：

```
y=1.3f;
```

该变量指定返回值字符串在视区中的垂直位置。

(2) 添加保护成员函数

添加 5 个保护成员函数：DrawFdbkObjects()，OutputInfo()，VertexData()，OutputFloatValue()和 ApplyFeedback()。

DrawFdbkObjects()函数负责物体的绘制和标记的插入，OutputInfo()函数负责输出反馈数组中的相关信息，VertexData()函数读取反馈数组中的一组数据，而 OutputFloatValue()函数则将浮点数转化为字符串并显示在视区指定的位置。现分别编辑如下：

```
void CSlctFdbkView::DrawFdbkObjects(GLenum mode)
{
    glPointSize(3.0f);
    glBegin (GL_POINTS);
        glNormal3f (0.0f, 0.0f, 1.0f);
        glVertex3f (0.5f, 0.0f, 0.0f);
    glEnd ();
    if (mode == GL_FEEDBACK) glPassThrough (1.0);
    glBegin (GL_POINTS);
        glVertex3f (10.0f, -10.0f, -10.0f);
    glEnd ();
    if (mode == GL_FEEDBACK) glPassThrough (2.0);
    glBegin (GL_POINTS);
        glNormal3f (0.0f, 0.0f, 1.0f);
        glVertex3f (0.5f, 0.5f, 0.0f);
    glEnd ();
}
```

很显然，第二个点将位于视景体之外。要注意，只有在反馈模式下才可调用 glPassThrough()函数。

```
void CSlctFdbkView::OutputInfo(GLint size, GLfloat * buffer)
{
    GLint count;
    GLfloat token;
    count = size;
    while (count)
    {
        token = buffer[size-count];
```

```
            count--;
            if (token == GL_PASS_THROUGH_TOKEN) count--;
            else if (token == GL_POINT_TOKEN)
                VertexData (size, &count, buffer);
            else if (token == GL_LINE_TOKEN)
            {
                VertexData (size, &count, buffer);
                VertexData (size, &count, buffer);
            }
            else if (token == GL_LINE_RESET_TOKEN)
            {
                VertexData (size, &count, buffer);
                VertexData (size, &count, buffer);
            }
            //    and so on ... ...
        }
}
```

该函数逐个读取整个反馈数组中的数据。size 参数为反馈数组中实际填写的数的个数。buffer 参数则为反馈数组。由 while 语句读取数据，此时只能处理点和线段，当然可以按上述方法添加代码以便完整地处理各类几何图元对象。

```
void CSlctFdbkView::VertexData(GLint size, GLint *count, GLfloat *buffer)
{
    int i;
    for (i = 0; i < 7; i++)
    {
        OutputFloatValue(y,buffer[size-(*count)]);
        *count = *count - 1;
        y -= 0.12f;
    }
}
```

该函数读取反馈数组中除标记以外的数据，此时由于顶点被光照但无纹理，因而共有 7 个数据。每次向视区输出一个数据之后，由 y-=0.12f;把当前光栅位置向下移 0.12 个单位。

```
void CSlctFdbkView::OutputFloatValue(GLfloat position,GLfloat value)
{
    wglUseFontBitmaps(wglGetCurrentDC(),0,256,1000);
    glListBase(1000);
    glRasterPos3f(-1.2f,position,0.0f);

    int     i, decimal, sign, precision = 7;
    char    *buffer;
    float source = value;
```

```
    buffer = _fcvt( source, precision, &decimal, &sign );
    if(sign = = 0)
        glCallLists(1,GL_UNSIGNED_BYTE," ");
    else
        glCallLists(1,GL_UNSIGNED_BYTE,"-");
    glCallLists(decimal,GL_BYTE,buffer);
    glCallLists(1,GL_UNSIGNED_BYTE,".");
    for(i=0;i<decimal;i++)buffer++;
    glCallLists(precision,GL_BYTE,buffer);
}
```

该函数利用_fcvt()函数将一个浮点数转化为一个字符串。由于它不直接转换小数点,故应利用其返回值作一些适当的处理。

```
void CSlctFdbkView::ApplyFeedback()
{
    GLfloat feedBuffer[1024];
    GLint size;
    glClearColor (0.0, 0.0, 0.0, 0.0);
    glClear(GL_COLOR_BUFFER_BIT);
    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);

    GLfloat matDif[] = {1.0f, 1.0f, 1.0f, 1.0f};
    GLfloat matSpe[] = {1.0f, 1.0f, 1.0f, 1.0f};
    glMaterialfv(GL_FRONT, GL_DIFFUSE, matDif);
    glMaterialfv(GL_FRONT, GL_SPECULAR,matSpe);
    glMaterialf(GL_FRONT, GL_SHININESS, 25.0f);

    DrawFdbkObjects(GL_RENDER);

    glFeedbackBuffer (1024, GL_3D_COLOR, feedBuffer);
    glRenderMode (GL_FEEDBACK);
    DrawFdbkObjects(GL_FEEDBACK);
    size = glRenderMode (GL_RENDER);
    //    适用于在视区内只输出两个点的信息之情况
    if(y<(1.3f-14.0f*0.12f)) y=1.3f;
        OutputInfo (size, feedBuffer);
        glDisable(GL_LIGHTING);
}
```

该函数说明了应用反馈模式的基本方法。在应用反馈模式之前,应先建立一个反馈数组feedBuffer,然后在绘图模式下绘制场景中的物体。接着进入反馈模式。由第十一条语句退出反馈模式并进入绘图模式。在退出时,返回写入反馈数组中数据的个数,并由 size 变量来记录。最后调用 OutputInfo()函数来处理反馈数据。

(3) 编辑 DrawScene()函数

在 switch(m _ modeControl){... ...} 中添加如下语句:

```
case Feedback:    ApplyFeedback();    break;
```

(4) 运行程序

再次编译连接并运行程序,执行菜单 Mode \ Feedback ,结果在视区内绘制了两个点,同时在左侧打印了两组相关数据,每组数据由 7 个组成,前 3 个为顶点的几何坐标(窗口坐标表示),它为窗口坐标表示,后 4 个数分别代表 R,G,B,A。如图 15-2 所示。

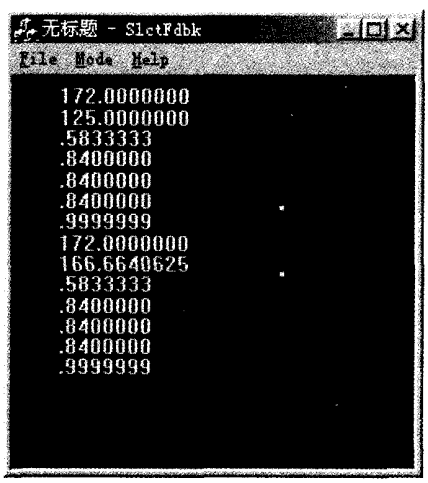

图 15-2 反馈模式

第 16 章　制作技术与应用实例

通过本书，我们已经学习了 OpenGL 最基本的编程技术。这些技术可以归为 3 类：物体建模技术、图像处理技术和人机交互技术。物体建模技术是最核心的技术，物体模型为图像处理和人机交互提供加工的源对象；图像处理对物体模型进行真实化，并对由物体模型所组成的二维画面进行操作；交互技术通过物体建模技术和图像处理技术提供用户控制输出画面的手段。

孤立地应用这些技术绝对不可能生成好的 OpenGL 软件产品。但是，对它们的有机结合也不是一朝一夕的事情。结合本身就是一个问题：结合什么，怎样结合？

一般来说，可以分成如下问题去探索：建立几何模型？建立物理模型？建立数学模型，并对问题进行数学描述？如何组织数据？建立人机交互的最小参数集？画面需要进一步美化？图像需要合成、分割和输出？

本章将描述上述问题并提供怎样解决这些问题的一般思路。具体地讲，本章将展示 6 个例程。第一个例程：说明建立数学模型是解决问题的重要途径，通过两个问题来描述，即，兔子、乌龟的相互追逐，C 曲线与分形；第二和第三个例程：介绍组织好数据是建立场景的关键，将展示两个例程，即，飘动的旗帜和滚动的西瓜；第四个例程：介绍如何将图像画到位图，如何将画面输出到外部文件（从中还可以学习怎样对画面进行艺术处理）；第五个例程：以类的形式介绍如何进行阴影制作（因而程序具有通用性）；第六个例程：牛头刨床急回机构的运动仿真。

16.1　建立数学模型是解决问题的重要途径

16.1.1　解决问题的方案

随着现代化生产实践和科学技术的不断发展，数学模型显得越来越重要。一谈到数学模型，首先想到的就是卫星发射、人口增长、军备竞赛、投资决策、市场平衡等问题，而往往忽视数学模型同样可以应用于图像制作特别是基于图像可视化的仿真模拟之中。建立问题的数学模型本身就是一种方法。事实上，数学模型是关于部分现实世界和为一种特殊目的而作的一个抽象、简化的结构。

数学模型的建立方法首先应该是观察。在建立模型前应对实际问题的背景有深刻的了解，进行全面的、深入细致的观察。明确所要解决的问题的目的和要求，并进行数据的收集整理和组织。其次应将问题进行简化。现实问题错综复杂，涉及面广，必须将问题简单化，即首先应抓住主要因素，在相对比较简单的情况下，理清变量之间的关系，建立相应的模型。有时候，一个模型需要假设。假设合理，则模型与实际问题就较吻合，如果假设不合理或过于简单，则需要修改模型。接着就应该建立模型。应分清变量的类型和使用什么样的数学工具；抓住问题的本质，简化变量之间的关系。最后应进行模型的检验与修改。模型的检验可以是基于数学工具的检验，也可以是可视化检验。可视化检验重点在于对模型中运动变量（或运动对象）的验证。

比如:有两只兔子和两只乌龟,分别位于4个角上,并可以相互追逐。假设第一只乌龟先移动,这样第一只兔子和第一只乌龟间可用斜线连接,第一只兔子应该在该斜线上追逐第一只乌龟。由于第一只兔子进行了移动,因而第二只乌龟和第一只兔子也可以用斜线来连接,并且第二只乌龟应该在此斜线上追逐第一只兔子,依次类推并如此往复。假设首先要弄清楚它们相互追逐的轨迹,应该怎样建立这个模型呢?又怎样用 OpenGL 来实现呢?

事实上,完成这一任务十分简单,只需将乌龟和兔子分别用一个点来代替,求新产生的斜线上乌龟和兔子的位置坐标即可。

又如,分形是一个应用前景十分广泛的领域。IBM 公司的 Benoit Mandebrot 首次提出了分形(Fractal)的几何概念,他在"The Fractal Geometry of Nature"一书中的开头一段指出:"为什么几何学常被认为枯燥而无味呢?其中的一个原因,就在于描述云彩、山峰、海岸线或树等物体的形状时所表现出的无能。云彩不是球状的,山峰不是圆锥形,海岸线也并非圆形。树皮也并不光滑,它们都不会像直线上的'旅行'那么轻松自如"。

在此之后,他指出,这些物体大多数是"相似于所有尺度的"。这是理解分形的简单方式,它们都有这样一个特点,即无论在多大的放大镜下观看,都保持相似的形状。大尺度的结构总是由无数个相似的小尺度结构组合而成。Mandelbrot 给出的一个典型例子就是海岸线:无论在飞机上、地面上,还是放大镜下,总会得到同样的形状。

通用分形的伪代码可以如下描述:第一,先画出大尺度下的物体;第二,将此大尺度的物体的各个部分用与其相似的小物体来代替。

最早发现的分形图形之一是康托雪花(Koch Snowflake)。事实上,它是由不断缩小的星形构成的。这个星形由两个等腰三角形构成,一个顶角向下,一个顶角向上,两底边平行,并对称放置。这个星形用直线构成,若将其已有直线均用一个带尖角的两条直线来代替就扩展出C曲线。下一小节将实现由C曲线构成的一种分形图。

16.1.2 乌龟和兔子的追逐、分形

现在,我们来完成程序设计,其步骤如下:

(1) 建立工程

建立一个名为 Fractal 的工程,并按第5章中的方法极小化 OpenGL。最后参考一下程序清单,看看极小化工作有什么不同之处。

(2) 增加枚举变量

在 FractalView.h 文件的头部增加枚举变量:

```
enum{RabtToTos, Fractal};
```

(3) 为 CFractalView 类添加保护成员变量

```
int   m_controlType;
```

初始化为:

```
m_controlType = RabtToTos;
```

(4) 添加保护成员函数

```
void        CreateFractals(GLfloat x1, GLfloat y1, GLfloat x2, GLfloat y2);
void        myFractal();
GLfloat     MoveY(GLfloat x1, GLfloat y1, GLfloat x2, GLfloat y2);
GLfloat     MoveX(GLfloat x1,GLfloat y1,GLfloat x2,GLfloat y2);
GLdouble    Distance(GLfloat x1,GLfloat y1,GLfloat x2,GLfloat y2);
void        Chase();
void        MoveThem(GLfloat * fa,GLfloat fb,GLfloat ft);
```

请按程序列表中相应函数进行编辑。

(5) 在 FractalView.cpp 文件的首部添加定义

```
#define PI ((float)3.14159265358979323846)
#define RADIANS(fAngle) ((float)(fAngle) * PI/180.0)
```

(6) 添加一个菜单

添加一个名为 Select 的菜单,即在属性对话框的 Caption 栏内加入 S&elect。增加以下菜单项,表 16-1 是它们的属性对话框中的 Caption 及 ID 号。

表 16-1

ID	Caption
IDM_RABBIT	&Rabbit and Tortoise
IDM_FRACTAL	F&ractal

请按程序列表中相应位置对其进行响应:

```
void CFractalView::OnRabbit()
{
    m_controlType = RabtToTos;
    Invalidate();
}

void CFractalView::OnUpdateRabbit(CCmdUI * pCmdUI)
{
    if( m_controlType == = RabtToTos)
        pCmdUI - > SetCheck(TRUE);
    else
        pCmdUI - > SetCheck(FALSE);
}

void CFractalView::OnFractal()
{
    m_controlType = Fractal;
    Invalidate();
}
```

```
void CFractalView::OnUpdateFractal(CCmdUI* pCmdUI)
{
    if( m_controlType==Fractal)
        pCmdUI->SetCheck(TRUE);
    else
        pCmdUI->SetCheck(FALSE);
}
```

(7) 编辑 DrawScene()函数

(8) 运行程序

编译连接运行程序。执行菜单 Select \ Rabbit and Tortoise,其结果如图 16-1 所示。执行菜单 Select \ Fractal,其结果如图 16-2 所示。

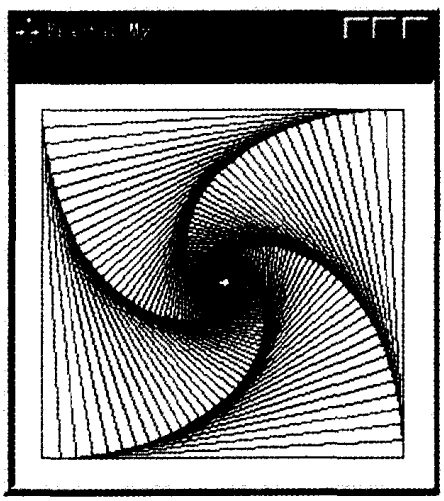

图 16-1 追逐图

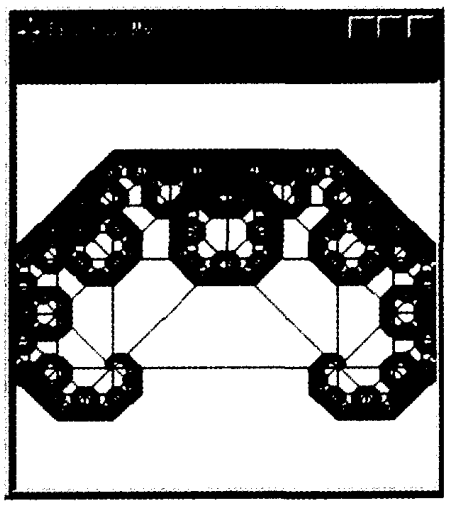

图 16-2 分形图

16.1.3 程序清单

FractalView.h 文件:

```
//////////////////////////////////////////////////////////////////////////////////////////
//////////////////////////                                  //////////////////////////
//////////////////////////          FractalView.h           //////////////////////////
//////////////////////////                                  //////////////////////////
//////////////////////////////////////////////////////////////////////////////////////////

#include "gl\gl.h"
#include "gl\glu.h"

enum{RabtToTos,Fractal};
```

```
#if !defined(AFX_FRACTALVIEW_H__19ED91CD_7FDF_11D2_83B3_0080C832FCF3__
            INCLUDED_)
#define AFX_FRACTALVIEW_H__19ED91CD_7FDF_11D2_83B3_0080C832FCF3__INCLUDED_

#if _MSC_VER >= 1000
#pragma once
#endif // _MSC_VER >= 1000

class CFractalView : public CView
{
protected: // create from serialization only
    CFractalView();
    DECLARE_DYNCREATE(CFractalView)

// Attributes
public:
    CFractalDoc* GetDocument();

    CClientDC     *m_pDC;
    CRect         m_oldRect;

// Operations
public:

// Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CFractalView)
    public:
    virtual void OnDraw(CDC* pDC); // overridden to draw this view
    virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
    protected:
    //}}AFX_VIRTUAL

// Implementation
public:
    virtual ~CFractalView();
#ifdef _DEBUG
    virtual void AssertValid() const;
    virtual void Dump(CDumpContext& dc) const;
#endif

protected:

    int   m_controlType;              //控制输出类型
```

```
// Generated message map functions
protected:
        void CreateFractals(GLfloat x1, GLfloat y1, GLfloat x2, GLfloat y2);
        void myFractal();
        GLfloat MoveY(GLfloat x1, GLfloat y1, GLfloat x2, GLfloat y2);
        GLfloat MoveX(GLfloat x1,GLfloat y1,GLfloat x2,GLfloat y2);
        GLdouble Distance(GLfloat x1,GLfloat y1,GLfloat x2,GLfloat y2);
        void Chase();
        void MoveThem(GLfloat * fa,GLfloat fb,GLfloat ft);
        void DrawScene();
        BOOL bSetupPixelFormat();
        void Init();
        //{{AFX_MSG(CFractalView)
        afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
        afx_msg void OnDestroy();
        afx_msg void OnSize(UINT nType, int cx, int cy);
        afx_msg void OnRabbit();
        afx_msg void OnUpdateRabbit(CCmdUI * pCmdUI);
        afx_msg void OnFractal();
        afx_msg void OnUpdateFractal(CCmdUI * pCmdUI);
        //}}AFX_MSG
        DECLARE_MESSAGE_MAP()
};

#ifndef _DEBUG // debug version in FractalView.cpp
inline CFractalDoc * CFractalView::GetDocument()
        { return (CFractalDoc * )m_pDocument; }
#endif

/////////////////////////////////////////////////////////////////////////////

//{{AFX_INSERT_LOCATION}}
// Microsoft Developer Studio will insert additional declarations immediately before the previous line.

#endif // !defined(AFX_FRACTALVIEW_H__19ED91CD_7FDF_11D2_83B3_0080C832FCF3__
                                                INCLUDED_)
```

FractalView.cpp 文件：

```
////////////////////////////////////////////////////////////////////////////////////////
//////////////////////////                                  //////////////////////////
//////////////////////////          FractalView.cap         //////////////////////////
//////////////////////////                                  //////////////////////////
////////////////////////////////////////////////////////////////////////////////////////
```

```
# include "stdafx.h"
# include "Fractal.h"

# include "FractalDoc.h"
# include "FractalView.h"

# include <math.h>

# define PI ((float)3.14159265358979323846)
# define RADIANS(fAngle) ((float)(fAngle) * PI/180.0)

# ifdef _DEBUG
# define new DEBUG_NEW
# undef THIS_FILE
static char THIS_FILE[] = __FILE__;
# endif

/////////////////////////////////////////////////////////////////////////////
// CFractalView

IMPLEMENT_DYNCREATE(CFractalView, CView)

BEGIN_MESSAGE_MAP(CFractalView, CView)
    //{{AFX_MSG_MAP(CFractalView)
    ON_WM_CREATE()
    ON_WM_DESTROY()
    ON_WM_SIZE()
    ON_COMMAND(IDM_RABBIT, OnRabbit)
    ON_UPDATE_COMMAND_UI(IDM_RABBIT, OnUpdateRabbit)
    ON_COMMAND(IDM_FRACTAL, OnFractal)
    ON_UPDATE_COMMAND_UI(IDM_FRACTAL, OnUpdateFractal)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CFractalView construction/destruction

CFractalView::CFractalView()
{
    // TODO: add construction code here
    m_pDC = NULL;
    m_controlType = RabtToTos;
}
```

```
CFractalView::~CFractalView()
{
}

BOOL CFractalView::PreCreateWindow(CREATESTRUCT& cs)
{
	// TODO: Modify the Window class or styles here by modifying
	// the CREATESTRUCT cs
	cs.style |= WS_CLIPCHILDREN | WS_CLIPSIBLINGS;
	return CView::PreCreateWindow(cs);
}

/////////////////////////////////////////////////////////////////////////////
// CFractalView drawing

void CFractalView::OnDraw(CDC* pDC)
{
	CFractalDoc* pDoc = GetDocument();
	ASSERT_VALID(pDoc);

	// TODO: add draw code for native data here
	DrawScene();
}

/////////////////////////////////////////////////////////////////////////////
// CFractalView diagnostics

#ifdef _DEBUG
void CFractalView::AssertValid() const
{
	CView::AssertValid();
}

void CFractalView::Dump(CDumpContext& dc) const
{
	CView::Dump(dc);
}

CFractalDoc* CFractalView::GetDocument() // non-debug version is inline
{
	ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(CFractalDoc)));
	return (CFractalDoc*)m_pDocument;
}
```

```
# endif //_DEBUG

/////////////////////////////////////////////////////////////////////////////
// CFractalView message handlers

void CFractalView::Init()
{
    PIXELFORMATDESCRIPTOR pfd;
    int         n;
    HGLRC       hrc;
    m_pDC = new CClientDC(this);
    ASSERT(m_pDC != NULL);
    if (!bSetupPixelFormat()) return;

    n = ::GetPixelFormat(m_pDC->GetSafeHdc());
    ::DescribePixelFormat(m_pDC->GetSafeHdc(), n, sizeof(pfd), &pfd);
    hrc = wglCreateContext(m_pDC->GetSafeHdc());
    wglMakeCurrent(m_pDC->GetSafeHdc(), hrc);

    GetClientRect(&m_oldRect);
    glClearDepth(1.0f);
    glEnable(GL_DEPTH_TEST);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

BOOL CFractalView::bSetupPixelFormat()
{
    static PIXELFORMATDESCRIPTOR pfd =
    {
      sizeof(PIXELFORMATDESCRIPTOR),
      1,
      PFD_DRAW_TO_WINDOW |
        PFD_SUPPORT_OPENGL,
      PFD_TYPE_RGBA
      24,
      0, 0, 0, 0, 0, 0,
      0,
      0,
      0,
      0, 0, 0, 0,
      32,
      0,
      0,
```

```
        PFD_MAIN_PLANE,
        0,
        0, 0, 0
    };
    int pixelformat;
    if ( (pixelformat = ChoosePixelFormat(m_pDC->GetSafeHdc(), &pfd)) == 0 )
    {
        MessageBox("ChoosePixelFormat failed");
        return FALSE;
    }
    if (SetPixelFormat(m_pDC->GetSafeHdc(), pixelformat, &pfd) == FALSE)
    {
        MessageBox("SetPixelFormat failed");
        return FALSE;
    }
    return TRUE;
}

int CFractalView::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    if (CView::OnCreate(lpCreateStruct) == -1)
        return -1;
    Init();
    return 0;
}

void CFractalView::OnDestroy()
{
    HGLRC hrc;
    hrc = ::wglGetCurrentContext();
    ::wglMakeCurrent(NULL, NULL);
    if (hrc) ::wglDeleteContext(hrc);
    if (m_pDC)
        delete m_pDC;
    CView::OnDestroy();
}

void CFractalView::OnSize(UINT nType, int cx, int cy)
{
    CView::OnSize(nType, cx, cy);
    if(cy > 0)
    {
        if((m_oldRect.right > cx) || (m_oldRect.bottom > cy))
        RedrawWindow();
```

```
        m_oldRect.right = cx;
        m_oldRect.bottom = cy;
        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
          glFrustum(-1.0,1.0,-1.0,1.0,3.0,7.0);
          glViewport(0, 0, cx, cy);
    }
}

void CFractalView::DrawScene()
{
    glClearColor(1.0f,1.0f,1.0f,1.0f);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glColor3f(1.0f, 0.0f, 0.0f);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glTranslatef(0.0f,0.0f,-4.5f);
    glColor3f(1.0f,0.0f,0.0f);
    switch(m_controlType)
    {
        case    RabtToTos:    Chase();    break;
        case    Fractal:      myFractal();    break;
    }
    glFinish();
}

void CFractalView::MoveThem(GLfloat* fa, GLfloat fb, GLfloat ft)
{
    *fa = (1-ft)*(*fa) + ft*fb;
}

void CFractalView::Chase()
{
    GLfloat    x1, y1;
    GLfloat    x2, y2;
    GLfloat    x3, y3;
    GLfloat    x4, y4;
    GLfloat    t=0.05;

    x1 = -1.3f;          y1 = -1.3f;
    x2 = 1.3f;           y2 = -1.3f;
    x3 = 1.3f;           y3 = 1.3f;
    x4 = -1.3f;          y4 = 1.3f;
    int   iColor=0;
```

```
    do
    {
        glColor3f((GLfloat)iColor/256.0 - 0.1f,0.3f,1.0f - (GLfloat)iColor/256.0);
        //标定位置
        glBegin(GL_LINE_LOOP);
            glVertex2f(x1,y1);
            glVertex2f(x2,y2);
            glVertex2f(x3,y3);
            glVertex2f(x4,y4);
            glVertex2f(x1,y1);
        glEnd();
        //追逐阶段
        MoveThem(&x1,x2,t);                    //第一只龟的移动:x
        MoveThem(&y1,y2,t);                    //第一只龟的移动:y
        MoveThem(&x2,x3,t);                    //第一只兔的移动:x
        MoveThem(&y2,y3,t);                    //第一只兔的移动:y
        MoveThem(&x3,x4,t);                    //第二只龟的移动:x
        MoveThem(&y3,y4,t);                    //第二只龟的移动:y
        MoveThem(&x4,x1,t);                    //第二只兔的移动:x
        MoveThem(&y4,y1,t);                    //第二只兔的移动:y
        if(iColor < 192)
            iColor += 16;
        else
            iColor += 2;
    }while(Distance(x1,y1,x2,y2) > 0.06);
}

GLdouble CFractalView::Distance(GLfloat x1, GLfloat y1, GLfloat x2, GLfloat y2)
{
    return sqrt((x2 - x1) * (x2 - x1) + (y2 - y1) * (y2 - y1));
}

void CFractalView::OnRabbit()
{
    m_controlType = RabtToTos;
    Invalidate();
}

void CFractalView::OnUpdateRabbit(CCmdUI * pCmdUI)
{
    if( m_controlType == RabtToTos)
        pCmdUI->SetCheck(TRUE);
    else
        pCmdUI->SetCheck(FALSE);
```

```
}

void CFractalView::OnFractal()
{
    m_controlType = Fractal;
    Invalidate();
}

void CFractalView::OnUpdateFractal(CCmdUI* pCmdUI)
{
    if( m_controlType == Fractal)
        pCmdUI->SetCheck(TRUE);
    else
        pCmdUI->SetCheck(FALSE);
}

GLfloat CFractalView::MoveX(GLfloat x1, GLfloat y1, GLfloat x2, GLfloat y2)
{
    GLfloat xMove;
    GLfloat fAngle;
    GLfloat fDist;
    GLfloat fRadius;
    GLfloat xShift;
    GLfloat yShift;
    fAngle = RADIANS(45.0);
    fDist = Distance(x1,y1,x2,y2);
    fRadius = (sqrt(2.0)/2.0) * fDist;
    //沿直线的移动:x 坐标
    xShift = cos(fAngle) * (x2 - x1);
    yShift = sin(fAngle) * (y2 - y1);
    xMove = fRadius/fDist * (xShift - yShift);
    return xMove;
}

GLfloat CFractalView::Movey(GLfloat x1, GLfloat y1, GLfloat x2, GLfloat y2)
{
    GLfloat yMove;
    GLfloat fAngle;
    GLfloat fDist;
    GLfloat fRadius;
    GLfloat xShift;
    GLfloat yShift;
    fAngle = RADIANS(45.0);
    fDist = Distance(x1,y1,x2,y2);
```

```
    fRadius = (sqrt(2.0)/2.0) * fDist;
    //沿直线的移动:y 坐标
    xShift = cos(fAngle) * (x2 - x1);
    yShift = sin(fAngle) * (y2 - y1);
    yMove = fRadius/fDist * (xShift + yShift);
    return yMove;
}

void CFractalView::myFractal()
{
    GLfloat     x1,y1,x2,y2;
    //分形区域的大小
    x1 = -0.8f;          y1 = -0.6f;
    x2 = 0.8f;           y2 = -0.6f;

    CreateFractals(x1,y1,x2,y2);
}

void CFractalView::CreateFractals(GLfloat x1, GLfloat y1, GLfloat x2, GLfloat y2)
{
    GLfloat fDistPt;
    GLfloat xSpike,ySpike;

    fDistPt = Distance(x1,y1,x2,y2);
    if(fDistPt <= 0.001)return;              //递归结束条件
    xSpike = x1 + Movex(x1,y1,x2,y2);
    ySpike = y1 + Movey(x1,y1,x2,y2);
    //创建分形:
    glBegin(GL_LINES);
        glVertex2f(x1,y1);
        glVertex2f(x2,y2);
        glVertex2f(x1,y1);
        glVertex2f(xSpike,ySpike);
        glVertex2f(xSpike,ySpike);
        glVertex2f(x2,y2);
    glEnd();

    CreateFractals(x1,y1,xSpike,ySpike);                    //递归调用
    CreateFractals(xSpike,ySpike,x2,y2);                    //递归调用
}
```

16.2 数据的组织:飘动的旗帜

16.2.1 解决问题的方案

第4章介绍了二维形状的制作,以及由二维的点线面及其拓扑结构构造三维形状的方法。第5章又介绍了怎样用矩阵堆栈来组织空间物体的位置关系。这些方法是很直接的,或者说,是基于人眼视觉系统的。因为,我们在日常生活中就是这样观察物体的,也是这样去考察物体的:物体是由多个面构成的,由线围成面,并由面构成体。事实上,问题并不是这么简单。请回忆一下,第7章中为什么盒子看起来并不是盒子。因为光照不正确,或者说,物体没有包含足够的信息。OpenGL只处理两种最小的基元,即几何顶点和像素。要建立良好的三维形体,必须处理好几何顶点,这里不仅仅指几何顶点位置坐标要正确,更重要的是,要为基片操作提供正确的、足够的操作信息,这些信息是:顶点的颜色、顶点的法线向量、顶点的纹理坐标等。顶点的纹理坐标决定了纹理图像如何贴到物体的表面。请思考这样一个问题:假设要将一个标签贴到圆柱上,应该怎样来确定构成圆柱的几何顶点的纹理坐标呢?所有这些问题本质上就是如何组织数据的问题。组织数据的核心问题就是几何模型的表达问题。

模型的表达是计算机图像处理的一个难点:第一,模型应包含物体的完整的信息,对一些简单物体这一点还比较容易满足,但对于复杂物体模型,这种表达将会变得很复杂而不易应用;第二,模型应有从具体到抽象的各种不同层次的表达方式。例如:一个具体的桌子可以用它的各部分的几何形状来描述,但"桌子"是具有相同功能的一类物体的总称。为了提高效率,增强用户对它的控制,我们不可能对所有具体的桌子都去建模,而是应该要建立一个一般的桌子模型。显然,仅仅用桌子的几何信息很难建立一个桌子的统一模型。因而,能否正确地表达模型和组织数据是建模成功的关键。

不像CAD、3D Studio Max等系统在建立物体模型时可以用体元(如,八面体、二十面体、球体、圆柱体、锥体、圆环面等)以及定义基元构造方式的布尔运算来完成,OpenGL只有一个几何基元可以采用:顶点。因而物体的拓扑结构是十分重要的,拓扑结构说明了物体的几何构成,一般可以用树、有向图等结构形式来表达。物体的顶点法线向量、颜色、纹理坐标既独立于几何拓扑结构又依赖于几何拓扑结构。解析表面的顶点法线向量一般各不相同,而刻板面的小面顶点法线向量是相同的,前者构成光滑的曲面,后者构成并不光滑的刻面物体。顶点的颜色描述物体的颜色属性,而纹理坐标则指示纹理图像应贴在物体的什么位置。一般来说,用OpenGL表达模型时应注意以下几个方面:

第一,模型表达应包含物体的完整信息,并具有唯一性。对于顶点,应包含其颜色、法线向量、纹理坐标,最后应包含该顶点的几何坐标。对于由点和线构成的面,应区分面的前后关系、填充模式、光照设定、纹理映射方式等。

第二,组织好物体几何基元的拓扑结构,实体与实体间的拓扑结构。这里,常采用两种方法来表达:其一为基于物体表面的表达;其二为基于实体模型的表达。基于实体模型的表达通常采用CSG树。它以体元为主要基元,要求基元的位置与几何参数可以用人机交互的方式输入。显然,CSG树隐藏了物体的表面信息,虽然紧凑,但当采用计算机图形学的算法时,却需要处理物体表面的几何信息。此时,应将构成物体的表面分割(称为有限单元化)成曲面小片,

并用几何顶点信息或参数来表达每个曲面，表达曲面片的交线等。三维世界由物体组成，物体表面由多个曲面构成，曲面与曲面之间的交线反映了物体表面的重要特性，这些几何元素可能处于模型的不同层次，应由数据结构清楚地表达出来。拓扑结构的表达不能有二义性。

第三，为了实际应用的方便，并减少在线计算，一些隐含的特征应在建立模型时计算好，例如，两个曲面的交线，虽然给定曲面后可以计算出来，也应作为几何元素存储于模型中。

第四，任何几何元素的表达都要使用一定的坐标系，为了表示的方便，物体系统各层次所使用的坐标系可以不同，因此，各层次间的坐标转换在建立模型时也应完成。一般地，最高层次的坐标为世界坐标，第二层次为物体坐标，第三层次为几何元素，如点、线、面等。在数据结构中，每个几何元素应用表结构给出其名字、性质，指向下一层次的指针及其坐标系数据。一般来说，每一层次最好使用同一种数据结构，这便于访问、查找、增删和重组等操作的实现。

第五，数据的组织要考虑方便的人机交互，以便于将用户的设想输入到计算机，其数据结构应符合计算机图形学的方法，以便于建立的模型可以用图像的形式显示出来。

16.2.2 飘动的旗帜

本节将制作旗帜飘动的动画，希望读者能够通过此例举一反三。将介绍如下技术：

- 如何组织顶点坐标、法线向量和纹理坐标；
- 如何对给定曲面实现有限单元化，如何组织曲面小片的结构；
- 如何实现由数据驱动的动画；
- 如何用纹理映射的方式来控制物体的亮度；
- 如何用键盘来实现人机交互；
- 程序的优化组织。

请按如下步骤完成：

(1) 建立工程

建立一个名为 Intractv 的工程，并按第 5 章中的方法极小化 OpenGL。最后请对照程序清单，修改极小化工作中的不同之处。

(2) 添加保护成员变量，并按构造函数中的对应语句进行初始化

```
GLint       m_framesNum;
GLint       m_widthX, m_widthY;
GLint       m_checkerSize;
GLint       m_contouring;
GLint       m_curFrame;
GLint       m_nextFrame;
GLenum      m_smooth;
GLenum      m_lighting;
GLenum      m_stepControl;
GLenum      m_rotate;
GLenum      m_clearMask;
GLenum      m_depth;
GLfloat     m_rotateAngle;
GLfloat     m_height;
```

```
BOOL            m_play;
struct  m_facet {
    float       color[3];
    float       normal[3];
};
struct m_coord {
    float  vertex[3];
    float       normal[3];
};
struct mesh {
    GLint       m_widthX,     m_widthY;
    GLint       numFacets;
    GLint       numCoords;
    GLint       numFrames;
    struct      m_coord * coords;
    struct      m_facet * facets;
} m_waveMesh;
```

(3)添加保护成员函数

```
void    KeyDown_R();
void    KeyDown_F();
void    KeyDown_K();
void    KeyDown_D();
void    KeyDown_L();
void    KeyDown_S();
void    KeyDown_T();
void    Wave();
void    InitTexture();
void    InitMaterials();
void    InitMesh();
```

请按程序列表中的相应位置处的语句进行编辑。

(4) 在文件 IntractvView.cpp 的首部分别添加:

```
#include <math.h>
#define     PI          3.14159265358979323846

GLubyte contourTexture1[] = {
    255, 255, 255, 255,
    255, 255, 255, 255,
    255, 255, 255, 255,
    127, 127, 127, 127,
};
GLubyte contourTexture2[] = {
    255, 255, 255, 255,
```

```
        255, 127, 127, 127,
        255, 127, 127, 127,
        255, 127, 127, 127,
    };
    #define GETm_coord(frame, x, y) (&(m_waveMesh.coords[frame * \
        m_waveMesh.numCoords + (x) + (y) * (m_waveMesh.m_widthX + 1)]))
    #define GETm_facet(frame, x, y) (&(m_waveMesh.facets[frame * \
        m_waveMesh.numFacets + (x) + (y) * m_waveMesh.m_widthX]))
```

(5) 响应 WM_KEYDOWN 消息和 WM_TIMER 消息,并按程序列表中的 OnKeyDown()函数及 OnTimer()函数进行编辑。

(6) 运行程序

编译连接并运行程序。一开始,程序绘制出一面旗帜,并不断地飘动,如图 16-3 所示。分别按下 D、F、K、L、R、S、T 各键,并进行各种组合,看看画面结果有什么变化。按下 D 键可以控制深度的测试;按下 F 键可以控制旗帜的绘制过程;按下 K 键则使旗帜停止飘动,再按一次则又飘动起来;按下 L 键可以取消光照或允许光照;按下 R 键则可以使旗帜旋转起来;按下 S 键则可以控制曲面是光滑的还是由刻板面组成的;按下 T 键则可以用纹理来控制旗帜的亮度。

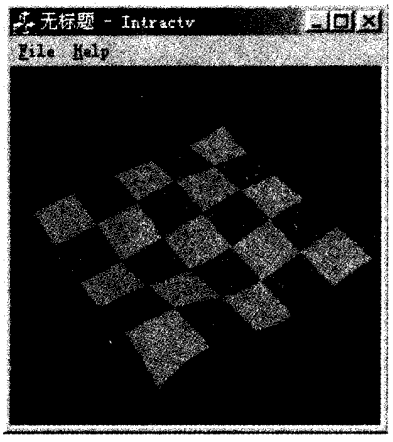

图 16-3 飘动的旗帜

16.2.3 程序清单

IntractvView.h 文件:

```
//////////////////////////////////////////////////////////////////////////////////////////
//////////////////////////                                  //////////////////////////
//////////////////////////          IntractvView.h          //////////////////////////
//////////////////////////                                  //////////////////////////
//////////////////////////////////////////////////////////////////////////////////////////
```

```
#include "gl\gl.h"
#include "gl\glu.h"

#if !defined(AFX_INTRACTVVIEW_H__5BAB0C82_7D5E_11D2_83B3_0080C832FCF3__
        INCLUDED_)
#define AFX_INTRACTVVIEW_H__5BAB0C82_7D5E_11D2_83B3_0080C832FCF3__INCLUDED_

#if _MSC_VER >= 1000
#pragma once
#endif // _MSC_VER >= 1000

class CIntractvView : public CView
{
protected: // create from serialization only
    CIntractvView();
    DECLARE_DYNCREATE(CIntractvView)

// Attributes
public:
    CIntractvDoc* GetDocument();

    CClientDC *m_pDC;
    CRect      m_oldRect;

// Operations
public:

// Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CIntractvView)
    public:
    virtual void OnDraw(CDC* pDC); // overridden to draw this view
    virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
    protected:
    //}}AFX_VIRTUAL

// Implementation
public:
    virtual ~CIntractvView();
#ifdef _DEBUG
    virtual void AssertValid() const;
    virtual void Dump(CDumpContext& dc) const;
#endif
```

```
protected:
    GLint m_framesNum;                  //数据驱动动画的帧更迭
    GLint m_widthX, m_widthY;           //旗帜的尺寸
    GLint m_checkerSize;                //旗帜纹理块大小
    GLint m_contouring;                 //纹理交换
    GLint m_curFrame;                   //帧交换:当前帧,数据驱动
    GLint m_nextFrame;                  //帧交换:下一帧,数据驱动

    GLenum      m_smooth;               //曲面光滑或是用小面来近似
    GLenum      m_lighting;             //光照与否
    GLenum      m_stepControl;          //数据流动还是数据静止
    GLenum      m_rotate;               //旗帜旋转与否
    GLenum      m_clearMask;            //清除颜色与深度缓存
    GLenum      m_depth;                //是否检测深度

    GLfloat m_rotateAngle;              //旋转角度
    GLfloat m_height;                   //旗帜飘动时掀起的振高

    BOOL m_play;                        //控制旗帜是飘动还是静止

    struct m_facet {
        float color[3];
        float normal[3];
    };                                  //小面数据的组织:颜色与法线

    struct m_coord {
        float vertex[3];
        float normal[3];
    };                                  //单个点数据组织:几何坐标与法线

    struct mesh {
        GLint m_widthX, m_widthY;
        GLint numFacets;
        GLint numCoords;
        GLint numFrames;
        struct m_coord * coords;
        struct m_facet * facets;
    } m_waveMesh;                       //网格小面的数据构成

// Generated message map functions
protected:
    void KeyDown_R();
    void KeyDown_F();
```

```
        void KeyDown_K();
        void KeyDown_D();
        void KeyDown_L();
        void KeyDown_S();
        void KeyDown_T();
        void Wave();
        void InitTexture();
        void InitMaterials();
        void InitMesh();
        void DrawScene();
        BOOL bSetupPixelFormat();
        void Init();
        //{{AFX_MSG(CIntractvView)
        afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
        afx_msg void OnDestroy();
        afx_msg void OnSize(UINT nType, int cx, int cy);
        afx_msg void OnTimer(UINT nIDEvent);
        afx_msg void OnKeyDown(UINT nChar, UINT nRepCnt, UINT nFlags);
        //}}AFX_MSG
        DECLARE_MESSAGE_MAP()
};

#ifndef _DEBUG // debug version in IntractvView.cpp
inline CIntractvDoc* CIntractvView::GetDocument()
{ return (CIntractvDoc*)m_pDocument; }
#endif

/////////////////////////////////////////////////////////////////////////////

//{{AFX_INSERT_LOCATION}}
// Microsoft Developer Studio will insert additional declarations immediately before the previous line.

#endif // !defined(AFX_INTRACTVVIEW_H__5BAB0C82_7D5E_11D2_83B3_0080C832FCF3__
                INCLUDED_)
```

IntractvView.cpp 文件：

```
////////////////////////////////////////////////////////////////////////////////////////////
//////////////////////////                                  //////////////////////////
//////////////////////////          IntractvView.cpp        //////////////////////////
//////////////////////////                                  //////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////

#include "stdafx.h"
#include "Intractv.h"
```

```
#include "IntractvDoc.h"
#include "IntractvView.h"

#include <math.h>

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

//////////////////////////////////////////////////////////////////////////////////////////

#define     PI     3.14159265358979323846

GLubyte contourTexture1[] = {
    255, 255, 255, 255,
    255, 255, 255, 255,
    255, 255, 255, 255,
    127, 127, 127, 127,
};    //纹理与亮度信息
GLubyte contourTexture2[] = {
    255, 255, 255, 255,
    255, 127, 127, 127,
    255, 127, 127, 127,
    255, 127, 127, 127,
};    //纹理与亮度信息

#define GETm_coord(frame, x, y) (&(m_waveMesh.coords[frame * \
        m_waveMesh.numCoords + (x) + (y) * (m_waveMesh.m_widthX + 1)]))
#define GETm_facet(frame, x, y) (&(m_waveMesh.facets[frame * \
        m_waveMesh.numFacets + (x) + (y) * m_waveMesh.m_widthX]))

//////////////////////////////////////////////////////////////////
// CIntractvView

IMPLEMENT_DYNCREATE(CIntractvView, CView)

BEGIN_MESSAGE_MAP(CIntractvView, CView)
    //{{AFX_MSG_MAP(CIntractvView)
    ON_WM_CREATE()
    ON_WM_DESTROY()
    ON_WM_SIZE()
```

```
    ON_WM_TIMER()
    ON_WM_KEYDOWN()
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CIntractvView construction/destruction

CIntractvView::CIntractvView()
{
    // TODO: add construction code here
    m_pDC = NULL;
    m_framesNum = 10;
    m_widthX = 10;
    m_widthY = 10;
    m_checkerSize = 2;
    m_height = 0.2;
    m_smooth = GL_TRUE;
    m_lighting = GL_TRUE;
    m_stepControl = GL_FALSE;
    m_rotate = GL_FALSE;
    m_clearMask = GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT;
    m_depth = GL_TRUE;
    m_play = TRUE;

    m_contouring = 0;
    m_curFrame = 0;
    m_nextFrame = 0;
    m_rotateAngle = 0.0f;
}

CIntractvView::~CIntractvView()
{
}

BOOL CIntractvView::PreCreateWindow(CREATESTRUCT& cs)
{
    // TODO: Modify the Window class or styles here by modifying
    // the CREATESTRUCT cs

    return CView::PreCreateWindow(cs);
}

/////////////////////////////////////////////////////////////////////////////
```

```
// CIntractvView drawing

void CIntractvView::OnDraw(CDC* pDC)
{
	CIntractvDoc* pDoc = GetDocument();
	ASSERT_VALID(pDoc);

	// TODO: add draw code for native data here
	DrawScene();
}

/////////////////////////////////////////////////////////////////////////////
// CIntractvView diagnostics

#ifdef _DEBUG
void CIntractvView::AssertValid() const
{
	CView::AssertValid();
}

void CIntractvView::Dump(CDumpContext& dc) const
{
	CView::Dump(dc);
}

CIntractvDoc* CIntractvView::GetDocument() // non-debug version is inline
{
	ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(CIntractvDoc)));
	return (CIntractvDoc*)m_pDocument;
}
#endif //_DEBUG

/////////////////////////////////////////////////////////////////////////////
// CIntractvView message handlers

void CIntractvView::Init()
{
	PIXELFORMATDESCRIPTOR pfd;
	int         n;
	HGLRC       hrc;
	m_pDC = new CClientDC(this);
	ASSERT(m_pDC != NULL);
	if (!bSetupPixelFormat()) return;
```

```
    n = ::GetPixelFormat(m_pDC->GetSafeHdc());
    ::DescribePixelFormat(m_pDC->GetSafeHdc(), n, sizeof(pfd), &pfd);
    hrc = wglCreateContext(m_pDC->GetSafeHdc());
    wglMakeCurrent(m_pDC->GetSafeHdc(), hrc);

    GetClientRect(&m_oldRect);
    glClearDepth(1.0f);
    glEnable(GL_DEPTH_TEST);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

BOOL CIntractvView::bSetupPixelFormat()
{
    static PIXELFORMATDESCRIPTOR pfd =
    {
      sizeof(PIXELFORMATDESCRIPTOR),
      1,
      PFD_DRAW_TO_WINDOW |
        PFD_SUPPORT_OPENGL |
          PFD_DOUBLEBUFFER,
      PFD_TYPE_RGBA,
      24,
      0, 0, 0, 0, 0, 0,
      0,
      0,
      0,
      0, 0, 0, 0,
      32,
      0,
      0,
      PFD_MAIN_PLANE,
      0,
      0, 0, 0
    };
    int pixelformat;
    if ( (pixelformat = ChoosePixelFormat(m_pDC->GetSafeHdc(), &pfd)) == 0 )
    {
      MessageBox("ChoosePixelFormat failed");
      return FALSE;
    }
    if (SetPixelFormat(m_pDC->GetSafeHdc(), pixelformat, &pfd) == FALSE)
    {
      MessageBox("SetPixelFormat failed");
```

```
        return FALSE;
    }
    return TRUE;
}

int CIntractvView::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    if (CView::OnCreate(lpCreateStruct) == -1)
        return -1;
    Init();
    glFrontFace(GL_CW);
    InitMaterials();
    InitTexture();
    InitMesh();
    SetTimer(1, 30, NULL);
    return 0;
}

void CIntractvView::OnDestroy()
{
    HGLRC hrc;
    if(m_play)KillTimer(1);
    hrc = ::wglGetCurrentContext();
    ::wglMakeCurrent(NULL, NULL);
    if (hrc)
      ::wglDeleteContext(hrc);
    if (m_pDC)
      delete m_pDC;
      CView::OnDestroy();
}

void CIntractvView::OnSize(UINT nType, int cx, int cy)
{
    CView::OnSize(nType, cx, cy);
    if(cy > 0)
    {
        if((m_oldRect.right > cx) || (m_oldRect.bottom > cy))
        RedrawWindow();
      m_oldRect.right = cx;
      m_oldRect.bottom = cy;
      glMatrixMode(GL_PROJECTION);
      glLoadIdentity();
        glFrustum(-1.0,1.0,-1.0,1.0,3.0,7.0);
        glViewport(0, 0, cx, cy);
```

```
void CIntractvView::DrawScene()
{
        glClearColor(0.0f,0.0f,0.0f,1.0f);
        glClear(m_clearMask);
        glColor3f(1.0f, 0.0f, 0.0f);

        glMatrixMode(GL_MODELVIEW);
        glLoadIdentity();
        glTranslatef(0.0f,0.1f, -4.5f);

        glScalef(2.0, 2.0, 2.0);
        glRotatef( -35.0, 1.0, 0.0, 0.0);
        glRotatef(35.0, 0.0, 0.0, 1.0);

        Wave();

        glFinish();
        SwapBuffers(wglGetCurrentDC());
}

void CIntractvView::OnTimer(UINT nIDEvent)
{
        Invalidate(FALSE);

        CView::OnTimer(nIDEvent);
}

void CIntractvView::InitMesh()
{
        struct m_coord * m_coord;
        struct m_facet * m_facet;
        float dp1[3], dp2[3];
        float * pt1, * pt2, * pt3;
        float angle, d, x, y;
        GLint numFacets, numCoords, frameNum;
        int           i, j;
        //旗帜的宽度与飘扬的高度
        m_waveMesh.m_widthX = m_widthX;
        m_waveMesh.m_widthY = m_widthY;
        m_waveMesh.numFrames = m_framesNum;
```

```
//曲面的细化控制:旗帜小面的个数,顶点数
numFacets = m_widthX * m_widthY;
numCoords = (m_widthX + 1) * (m_widthY + 1);

m_waveMesh.numCoords = numCoords;
m_waveMesh.numFacets = numFacets;
//顶点信息内存分配
m_waveMesh.coords = (struct m_coord *)malloc(
                m_framesNum * numCoords * sizeof(struct m_coord));
//小面拓扑信息内存分配
m_waveMesh.facets = (struct m_facet *)malloc(
                m_framesNum * numFacets * sizeof(struct m_facet));

if (m_waveMesh.coords == NULL || m_waveMesh.facets == NULL)
{
    MessageBox("Out of Memory!");
    return;
}
//数据驱动机制的建立:数据流结构
//曲面的有限单元化
for (frameNum = 0; frameNum < m_framesNum; frameNum++)
{
    for (i = 0; i <= m_widthX; i++)
    {
        x = i / (float)m_widthX;
        for (j = 0; j <= m_widthY; j++)
        {
            y = j / (float)m_widthY;

            d = sqrt(x * x + y * y);
            if (d == 0.0)     d = 0.0001;

            angle = 2 * PI * d + (2 * PI / m_framesNum * frameNum);

            m_coord = GETm_coord(frameNum, i, j);
            //顶点坐标
            m_coord->vertex[0] = x - 0.5;              //X
            m_coord->vertex[1] = y - 0.5;              //Y
            m_coord->vertex[2] = (m_height - m_height * d) * cos(angle); //Z
            //法线
            m_coord->normal[0] = -(m_height / d) * x * ((1 - d) * 2 * PI *
        sin(angle) + cos(angle));
            m_coord->normal[1] = -(m_height / d) * y * ((1 - d) * 2 * PI *
        sin(angle) + cos(angle));
```

```
            m_coord->normal[2] = -1;
            //法线归一化
            d = 1.0 / sqrt(m_coord->normal[0] * m_coord->normal[0] +
        m_coord->normal[1] * m_coord->normal[1] + 1);
            m_coord->normal[0] *= d;
            m_coord->normal[1] *= d;
            m_coord->normal[2] *= d;
            }
}
for (i = 0; i < m_widthX; i++)
{
       for (j = 0; j < m_widthY; j++)
       {
            m_facet = GETm_facet(frameNum, i, j);              //获取小面的拓扑信息
            //赋给小面颜色:小面被分成棋块状
            if (((i/m_checkerSize)%2)^(j/m_checkerSize)%2)
            {
                    m_facet->color[0] = 1.0;
                    m_facet->color[1] = 0.1;
                    m_facet->color[2] = 0.1;
            }
            else
            {
                m_facet->color[0] = 0.2;
                m_facet->color[1] = 1.0;
                m_facet->color[2] = 0.2;
            }
            //获取小面顶点坐标
            pt1 = GETm_coord(frameNum, i, j)->vertex;
            pt2 = GETm_coord(frameNum, i, j+1)->vertex;
            pt3 = GETm_coord(frameNum, i+1, j+1)->vertex;
            //计算小面法线
            dp1[0] = pt2[0] - pt1[0];
            dp1[1] = pt2[1] - pt1[1];
            dp1[2] = pt2[2] - pt1[2];

            dp2[0] = pt3[0] - pt2[0];
            dp2[1] = pt3[1] - pt2[1];
            dp2[2] = pt3[2] - pt2[2];

            m_facet->normal[0] = dp1[1] * dp2[2] - dp1[2] * dp2[1];
            m_facet->normal[1] = dp1[2] * dp2[0] - dp1[0] * dp2[2];
            m_facet->normal[2] = dp1[0] * dp2[1] - dp1[1] * dp2[0];
            //法线归一化
```

```
                    d = 1.0 / sqrt(m_facet->normal[0] * m_facet->normal[0] +
                m_facet->normal[1] * m_facet->normal[1] +
                m_facet->normal[2] * m_facet->normal[2]);

                    m_facet->normal[0] *= d;
                    m_facet->normal[1] *= d;
                    m_facet->normal[2] *= d;
                }
            }
        }
}

void CIntractvView::InitMaterials()
{
        float ambient[] = {0.1, 0.1, 0.1, 1.0};
        float diffuse[] = {0.5, 1.0, 1.0, 1.0};
        float position[] = {4.0, 6.0, 10.0, 0.0};
        float front_mat_shininess[] = {60.0};
        float front_mat_specular[] = {0.2, 0.2, 0.2, 1.0};
        float front_mat_diffuse[] = {0.5, 0.28, 0.38, 1.0};
        float back_mat_shininess[] = {60.0};
        float back_mat_specular[] = {0.5, 0.5, 0.2, 1.0};
        float back_mat_diffuse[] = {1.0, 1.0, 0.2, 1.0};
        float model_ambient[] = {1.0, 1.0, 1.0, 1.0};
        float model_twoside[] = {GL_TRUE};
        //光照与材质
        glLightfv(GL_LIGHT0, GL_AMBIENT, ambient);
        glLightfv(GL_LIGHT0, GL_DIFFUSE, diffuse);
        glLightfv(GL_LIGHT0, GL_POSITION, position);
        glLightModelfv(GL_LIGHT_MODEL_AMBIENT, model_ambient);
        glLightModelfv(GL_LIGHT_MODEL_TWO_SIDE, model_twoside);

        glEnable(GL_LIGHTING);
        glEnable(GL_LIGHT0);

        glMaterialfv(GL_FRONT, GL_SHININESS, front_mat_shininess);
        glMaterialfv(GL_FRONT, GL_SPECULAR, front_mat_specular);
        glMaterialfv(GL_FRONT, GL_DIFFUSE, front_mat_diffuse);
        glMaterialfv(GL_BACK, GL_SHININESS, back_mat_shininess);
        glMaterialfv(GL_BACK, GL_SPECULAR, back_mat_specular);
        glMaterialfv(GL_BACK, GL_DIFFUSE, back_mat_diffuse);
        //颜色材质
        glColorMaterial(GL_FRONT_AND_BACK, GL_DIFFUSE);
        glEnable(GL_COLOR_MATERIAL);
```

```
}

void CIntractvView::InitTexture()
{
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
    glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
}

void CIntractvView::Wave()
{
    struct m_coord *          m_coord;
    struct m_facet *          m_facet;
    float * lastColor;
    float * thisColor;
    GLint i, j;

    if (m_nextFrame || ! m_stepControl)
    {
        m_curFrame++;      //动画控制参数
    }
    if (m_curFrame >= m_waveMesh.numFrames)
    {
        m_curFrame = 0;
    }

    if ((m_nextFrame || ! m_stepControl) && m_rotate)
    {
        m_rotateAngle += 5.0f;
        if(m_rotateAngle > 360.0f) m_rotateAngle = 0.0f;
        glRotatef(m_rotateAngle, 0.0f, 0.0f, 1.0f);
    }
    m_nextFrame = 0;
    //数据的流动:旗帜小面的重组
    for (i = 0; i < m_waveMesh.m_widthX; i++)
    {
        glBegin(GL_QUAD_STRIP);
        lastColor = NULL;
        for (j = 0; j < m_waveMesh.m_widthY; j++)
        {
            m_facet = GETm_facet(m_curFrame, i, j);          //获取即将计算的小面信息
            //是否光照与光顺处理
```

```
if (! m_smooth && m_lighting)
{
    glNormal3fv(m_facet->normal);
}
if (m_lighting)
{
    thisColor = m_facet->color;
    glColor3fv(m_facet->color);
}
//颜色流动,获取顶点几何等信息
if (! lastColor || (thisColor[0] != lastColor[0] && m_smooth))
{
    if (lastColor)
    {
        glEnd();
        glBegin(GL_QUAD_STRIP);
    }
    m_coord = GETm_coord(m_curFrame, i, j);
    if (m_smooth && m_lighting)
    {
        glNormal3fv(m_coord->normal);
    }
    glVertex3fv(m_coord->vertex);
    m_coord = GETm_coord(m_curFrame, i+1, j);
    if (m_smooth && m_lighting)
    {
        glNormal3fv(m_coord->normal);
    }
    glVertex3fv(m_coord->vertex);
}
m_coord = GETm_coord(m_curFrame, i, j+1);
if (m_smooth && m_lighting)
{
    glNormal3fv(m_coord->normal);
}
glVertex3fv(m_coord->vertex);

m_coord = GETm_coord(m_curFrame, i+1, j+1);
if (m_smooth && m_lighting)
{
    glNormal3fv(m_coord->normal);
}
glVertex3fv(m_coord->vertex);
lastColor = thisColor;
```

```
        glEnd();


void CIntractvView::OnKeyDown(UINT nChar, UINT nRepCnt, UINT nFlags)
{
    switch(nChar)
    {
        case 'T':       KeyDown_T();    break;
        case 'S':       KeyDown_S();    break;
        case 'L':       KeyDown_L();    break;
        case 'D':       KeyDown_D();    break;
        case 'K':       KeyDown_K();    break;
        case 'R':       KeyDown_R();    break;
        case 'F':       KeyDown_F();    break;
    }
    CView::OnKeyDown(nChar, nRepCnt, nFlags);
}

void CIntractvView::KeyDown_T()
{
    //纹理控制
    m_contouring++;
    if (m_contouring == 1)
    {
      static GLfloat map[4] = {0, 0, 20, 0};

      glTexImage2D(GL_TEXTURE_2D, 0, 3, 4, 4, 0, GL_LUMINANCE,
                   GL_UNSIGNED_BYTE, (GLvoid *)contourTexture1);
      glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_OBJECT_LINEAR);
      glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_OBJECT_LINEAR);
      glTexGenfv(GL_S, GL_OBJECT_PLANE, map);
      glTexGenfv(GL_T, GL_OBJECT_PLANE, map);
      glEnable(GL_TEXTURE_2D);
      glEnable(GL_TEXTURE_GEN_S);
      glEnable(GL_TEXTURE_GEN_T);
    }
    else if (m_contouring == 2)
    {
      static GLfloat map[4] = {0, 0, 20, 0};
      glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_EYE_LINEAR);
      glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_EYE_LINEAR);
```

```
            glPushMatrix();
                glMatrixMode(GL_MODELVIEW);
                    glLoadIdentity();
                    glTexGenfv(GL_S, GL_EYE_PLANE, map);
                    glTexGenfv(GL_T, GL_EYE_PLANE, map);
            glPopMatrix();
        }
        else
            {
            m_contouring = 0;
            glDisable(GL_TEXTURE_GEN_S);
            glDisable(GL_TEXTURE_GEN_T);
            glDisable(GL_TEXTURE_2D);
        }
}

void CIntractvView::KeyDown_S()
{
        //光顺控制
        m_smooth = ! m_smooth;
        if (m_smooth) {
            glShadeModel(GL_SMOOTH);
        } else {
            glShadeModel(GL_FLAT);
        }
}

void CIntractvView::KeyDown_L()
{
        //光照控制
        m_lighting = ! m_lighting;
        if (m_lighting) {
            glEnable(GL_LIGHTING);
            glEnable(GL_LIGHT0);
            glEnable(GL_COLOR_MATERIAL);
        }
        else
        {
            glDisable(GL_LIGHTING);
            glDisable(GL_LIGHT0);
            glDisable(GL_COLOR_MATERIAL);
        }
}
```

```
void CIntractvView::KeyDown_D()
{
    //深度测试控制
    m_depth = ! m_depth;
    if (m_depth)
    {
      glEnable(GL_DEPTH_TEST);
      m_clearMask |= GL_DEPTH_BUFFER_BIT;
    }
    else
    {
      glDisable(GL_DEPTH_TEST);
      m_clearMask &= ~GL_DEPTH_BUFFER_BIT;
    }
}

void CIntractvView::KeyDown_K()
{
    m_stepControl = ! m_stepControl;
    m_play = ! m_play;

    if (m_play)
        SetTimer(1, 30, NULL);
    else
        KillTimer(1);
}

void CIntractvView::KeyDown_F()
{
    //数据驱动机制的建立
    if (m_stepControl) m_nextFrame = 1;
}

void CIntractvView::KeyDown_R()
{
    //运动控制参数
    m_rotate = ! m_rotate;
}
```

16.3 数据的组织:滚动的西瓜

16.3.1 解决问题的方案

事实上,要建立良好的几何模型,并对模型进行表达,还需要很多计算机图形学和数学算法。其中,人的思维是最主要的。一个观察力敏锐的人,就像雕塑家一样,对物体的几何模型有一种特别的感觉。其实,用计算机来创建画面也需要这种感觉:源于观察、实践、思考和创造。以 3D Studio MAX 三维动画制作软件为例,为什么有的人能够制作出表现力非凡的画面,而有的人的作品却缺乏生气?

用计算机来创作三维画面,事实上就是一种美的创造。把自己的想象力和创造力外延于作品是一种表现的力量。通过什么来与 OpenGL 相结合呢? 唯一的道路就是编程。

比如,怎样来创作一个球? 球是曲面,按有限单元化的思想就是将它进行曲面小片化。地理位置是通过经纬线来标定的,一个球同样可以按这种方法来划分。经线和纬线的交点就是我们编程时要利用的几何顶点。相邻纬线之间的带状表面可以通过四边形片来构成。这就会遇到一个问题:南极和北极怎么处理? 可以通过三角形扇来完成两极表面的构造。法线向量则很好办,球在每一点的法线向量均过球心,指向观察者。纹理坐标则值得考究。

这是一种就事论事的解决问题的方法! 还有没有制作球的其他方法? 能不能从一个球削出一个规则的正 20 面体?

将球削出一个正 20 面体,可以认为这个球就是用 20 个面来近似的。现在,对于每个三角形,计算每条边的中点,中点和中点进行连线。这样一个三角形小面就用 4 个小的三角形来代替。最后把新生成的顶点坐标所表示的向量进行单位化,并将此向量乘上球的半径,这相当于将新增加的顶点拉到球面上。其结果是这个球不再是用 20 个面来近似而是用 80 个面来逼近,当然效果要好得多。如此细分下去,直到满足精度要求为止。事实上再细分一次就足够精确了。这就是一种算法,一种创建力或一种思想。很显然,用三角形小面来逼近球面时,不需要处理南极和北极的特殊情况。此时也不存在两极,每个小面均处于对等状态,不存在离赤道越近小面越大的毛病。

现在来制作一个西瓜,假定这个西瓜正好是一个球状。想一想西瓜皮:青黑相间,有经线状,有两极。这就决定了我们应该采用经纬线划分的方法来有限单元化这个球,因为同一条经线上的纹理或颜色应该大致相同,这就便于程序实现。

16.3.2 滚动的西瓜

在该例程中,将学习两方面的内容:第一,进一步学习数据的组织方法;第二,学习 OpenGL 的 glDrawElements()函数、glVertexPointer()函数、glNormalPointer()函数及 glColorPointer()函数等的用法。

现在,请按如下步骤完成编程:

(1) 建立工程

建立一个名为 Model 的工程,并按第 5 章中的方法极小化 OpenGL。最后应参考一下程序清单,看看是否有不同之处。

(2) 在 ModelView.h 文件的头部添加定义

```
#define PI ((float)3.14159265358979323846)

#define COMPUTE_SPHERE_VERTICES(rings, sections) \
    (((rings) + 1) * (sections) + 2)

#define COMPUTE_SPHERE_TRIANGLES(rings, sections) \
(((rings) + 1) * (sections) * 2)
#define RINGS          50
#define SECTIONS       50

#define SVERTICES COMPUTE_SPHERE_VERTICES(RINGS, SECTIONS)
#define STRIANGLES COMPUTE_SPHERE_TRIANGLES(RINGS, SECTIONS)

#define RGB_COLOR(red, green, blue) \
(((DWORD)(BYTE)(red) << 0) | \
((DWORD)(BYTE)(green) << 8) | \
((DWORD)(BYTE)(blue) << 16)

#define FRANDOM(x) (((float)rand() / RAND_MAX) * (x))
#define DROT           10.0f
#define NLISTS         3
```

(3) 添加保护成员变量

```
typedef struct
{
    float x, y, z;              //点
    float fNx, fNy, fNz;        //法线
    DWORD dwColor;
} VERTEX;

typedef struct
{
    int iV1;
    int iV2;
    int iV3;
} TRIANGLE;

typedef struct _SPHERE
{
    GLfloat xr, yr, zr;
    GLfloat fDxr, fDyr, fDzr;
} SPHERE;
```

```
SPHERE          m_sphere;
BOOL            m_fDisplayList;
int             m_iCurrentMode;
GLint           m_iLists[NLISTS];
DWORD           m_dwTotalMillis;
long            m_iTotalTriangles;

VERTEX          m_vtxSphere[SVERTICES];

TRIANGLE        m_triSphere[STRIANGLES];

BOOL            m_play;
BOOL            m_controlMode;
GLenum          m_selectMode;
```

(4) 添加保护成员函数,并按构造函数进行初始化

```
void    Key_S();
void    Key_Space();
void    ResetTotals();
void    PlayBall();
void    Init_Sphere_Data();
void    Vertex(int iVert);
void    UpdateSphere();
void    InitSphere();
void    GenerateSphere(float fRadius, int iRings, int iSections,
            VERTEX * pvtx, TRIANGLE * ptri);
void    DrawStrips();
void    DrawTriangles();
void    DrawVertexArray();
```

请自己按程序清单进行编辑。

(5) 响应 WM_KEYDOWN 消息和 WM_TIMER 消息,并按程序列表中的OnKeyDown()函数及 OnTimer()函数进行编辑。

(6) 增加文件包含

在 ModelView.cpp 文件的头部添加:

```
#include <math.h>
```

(7) 编辑 DrawScene()函数

(8) 运行程序

编译连接并运行程序。可见,一个西瓜正在滚动,如图 16-4。按下空格键,西瓜停止滚动,再按一次,又滚动了起来。按下 S 键,西瓜变成了一个线框球,如图 16-5。再按一次又变成了西瓜。

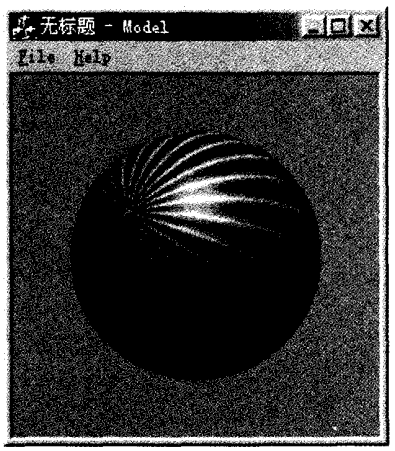

图 16-4　滚动的西瓜

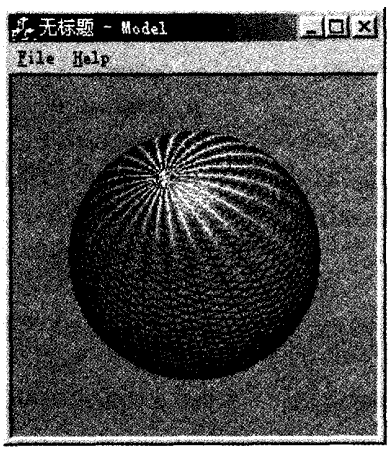

图 16-5　西瓜的构成

16.3.3　程序清单

ModelView.h 文件：

```
//////////////////////////////////////////////////////////////////////////////////////////
//////////////////////////                                  //////////////////////////
//////////////////////////          ModelView.h             //////////////////////////
//////////////////////////                                  //////////////////////////
//////////////////////////////////////////////////////////////////////////////////////////

# include "gl \ gl.h"
# include "gl \ glu.h"

# define    PI          ((float)3.14159265358979323846)

# define    COMPUTE_SPHERE_VERTICES(rings, sections) (((rings) + 1) * (sections) + 2)
# define    COMPUTE_SPHERE_TRIANGLES(rings, sections) (((rings) + 1) * (sections) * 2)

# define    RINGS           50
# define    SECTIONS        50

# define    SVERTICES       COMPUTE_SPHERE_VERTICES(RINGS, SECTIONS)
# define    STRIANGLES COMPUTE_SPHERE_TRIANGLES(RINGS, SECTIONS)

# define    RGB_COLOR(red, green, blue) \
          (((DWORD)(BYTE)(red) < < 0) | \
          ((DWORD)(BYTE)(green) < < 8) | \
          ((DWORD)(BYTE)(blue) < < 16))
```

```
#define     FRANDOM(x) (((float)rand() / RAND_MAX) * (x))
#define     DROT 10.0f
#define     NLISTS      3

//////////////////////////////////////////////
#if ! defined(AFX_MODELVIEW_H__3009D04D_7F26_11D2_83B3_0080C832FCF3__INCLUDED_)
#define AFX_MODELVIEW_H__3009D04D_7F26_11D2_83B3_0080C832FCF3__INCLUDED_

#if _MSC_VER >= 1000
#pragma once
#endif // _MSC_VER >= 1000

class CModelView : public CView
{
protected: // create from serialization only
        CModelView();
        DECLARE_DYNCREATE(CModelView)

// Attributes
public:
        CModelDoc* GetDocument();

        CClientDC     *m_pDC;
        CRect         m_oldRect;

// Operations
public:

// Overrides
        // ClassWizard generated virtual function overrides
        //{{AFX_VIRTUAL(CModelView)
        public:
        virtual void OnDraw(CDC* pDC); // overridden to draw this view
        virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
        protected:
        //}}AFX_VIRTUAL

// Implementation
public:
        virtual ~CModelView();
#ifdef _DEBUG
        virtual void AssertValid() const;
        virtual void Dump(CDumpContext& dc) const;
```

```
#endif

protected:
    typedef struct
    {
        float       x, y, z;                    //点
        float       fNx, fNy, fNz;              //法线
        DWORD dwColor;                          //颜色索引值
    } VERTEX;

    typedef struct
    {
        int     iV1;
        int     iV2;
        int     iV3;
    } TRIANGLE;                                 //三角形小面的顶点构成几何拓扑结构

    typedef struct _SPHERE
    {
        GLfloat     xr, yr, zr;
        GLfloat     fDXr, fDYr, fDZr;
    } SPHERE;                                   //球上一点的几何信息

    SPHERE      m_sphere;                       //顶点信息

    BOOL        m_fDisplayList;                 //显示列表
    int         m_iCurrentMode;                 //当前模式
    GLint       m_iLists[NLISTS];               //显示列表序列
    DWORD       m_dwTotalMillis;                //统计时间
    long        m_iTotalTriangles;              //统计已绘制三角形个数

    VERTEX      m_vtxSphere[SVERTICES];         //除两极外的顶点数据
    TRIANGLE    m_triSphere[STRIANGLES];        //两极顶点数据

    BOOL        m_play;                         //运动控制
    BOOL        m_controlMode;                  //模式控制
    GLenum      m_selectMode;                   //选择控制

// Generated message map functions
protected:
    void Key_S();
    void Key_Space();
    void ResetTotals();
    void PlayBall();
```

```
    void Init_Sphere_Data();
    void Vertex(int iVert);
    void UpdateSphere();
    void InitSphere();
    void GenerateSphere(float fRadius, int iRings, int iSections,
                        VERTEX * pvtx, TRIANGLE * ptri);
    void DrawStrips();
    void DrawTriangles();
    void DrawVertexArray();
    void DrawScene();
    BOOL bSetupPixelFormat();
    void Init();
    //{{AFX_MSG(CModelView)
    afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
    afx_msg void OnDestroy();
    afx_msg void OnSize(UINT nType, int cx, int cy);
    afx_msg void OnTimer(UINT nIDEvent);
    afx_msg void OnKeyDown(UINT nChar, UINT nRepCnt, UINT nFlags);
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};

#ifndef _DEBUG // debug version in ModelView.cpp
inline CModelDoc* CModelView::GetDocument()
   { return (CModelDoc*)m_pDocument; }
#endif

/////////////////////////////////////////////////////////////////////////////

//{{AFX_INSERT_LOCATION}}
// Microsoft Developer Studio will insert additional declarations immediately before the previous line.

#endif // !defined(AFX_MODELVIEW_H__3009D04D_7F26_11D2_83B3_0080C832FCF3__
       INCLUDED_)
```

ModelView.cpp 文件：

```
////////////////////////////////////////////////////////////////////////////////////////////
//////////////////////////                                  //////////////////////////
//////////////////////////          ModelView.cpp           //////////////////////////
//////////////////////////                                  //////////////////////////
////////////////////////////////////////////////////////////////////////////////////////

#include "stdafx.h"
#include "Model.h"
```

```
#include "ModelDoc.h"
#include "ModelView.h"

#include <math.h>

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
// CModelView

IMPLEMENT_DYNCREATE(CModelView, CView)

BEGIN_MESSAGE_MAP(CModelView, CView)
    //{{AFX_MSG_MAP(CModelView)
    ON_WM_CREATE()
    ON_WM_DESTROY()
    ON_WM_SIZE()
    ON_WM_TIMER()
    ON_WM_KEYDOWN()
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CModelView construction/destruction

CModelView::CModelView()
{
    // TODO: add construction code here
    m_pDC = NULL;
    m_fDisplayList = TRUE;
    m_iCurrentMode = 0;
    m_dwTotalMillis = 0;
    m_iTotalTriangles = 0;

    m_play = TRUE;
    m_controlMode = TRUE;
    m_selectMode = GL_FILL;
}
```

```
CModelView::~CModelView()
{
}

BOOL CModelView::PreCreateWindow(CREATESTRUCT& cs)
{
	// TODO: Modify the Window class or styles here by modifying
	// the CREATESTRUCT cs
	cs.style |= WS_CLIPCHILDREN | WS_CLIPSIBLINGS;
	return CView::PreCreateWindow(cs);
}

////////////////////////////////////////////////////////////////////////////
// CModelView drawing

void CModelView::OnDraw(CDC* pDC)
{
	CModelDoc* pDoc = GetDocument();
	ASSERT_VALID(pDoc);

	DrawScene();
}

////////////////////////////////////////////////////////////////////////////
// CModelView diagnostics

#ifdef _DEBUG
void CModelView::AssertValid() const
{
	CView::AssertValid();
}

void CModelView::Dump(CDumpContext& dc) const
{
	CView::Dump(dc);
}

CModelDoc* CModelView::GetDocument() // non-debug version is inline
{
	ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(CModelDoc)));
	return (CModelDoc*)m_pDocument;
}
#endif //_DEBUG
```

```
/////////////////////////////////////////////////////////////////////////////
// CModelView message handlers

void CModelView::Init()
{
    PIXELFORMATDESCRIPTOR pfd;
    int         n;
    HGLRC       hrc;
    m_pDC = new CClientDC(this);
    ASSERT(m_pDC != NULL);
    if (!bSetupPixelFormat())
      return;

    n = ::GetPixelFormat(m_pDC->GetSafeHdc());
    ::DescribePixelFormat(m_pDC->GetSafeHdc(), n, sizeof(pfd), &pfd);
    hrc = wglCreateContext(m_pDC->GetSafeHdc());
    wglMakeCurrent(m_pDC->GetSafeHdc(), hrc);

    GetClientRect(&m_oldRect);
    glClearDepth(1.0f);
    glEnable(GL_DEPTH_TEST);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

BOOL CModelView::bSetupPixelFormat()
{
    static PIXELFORMATDESCRIPTOR pfd =
    {
      sizeof(PIXELFORMATDESCRIPTOR),
      1,
      PFD_DRAW_TO_WINDOW |
        PFD_SUPPORT_OPENGL |
          PFD_DOUBLEBUFFER,
      PFD_TYPE_RGBA,
      24,
      0, 0, 0, 0, 0, 0,
      0,
      0,
      0,
      0, 0, 0, 0,
      32,
      0,
      0,
```

```
        PFD_MAIN_PLANE,
        0,
        0, 0, 0
    };
    int pixelformat;
    if ( (pixelformat = ChoosePixelFormat(m_pDC->GetSafeHdc(), &pfd)) == 0 )
    {
        MessageBox("ChoosePixelFormat failed");
        return FALSE;
    }
    if (SetPixelFormat(m_pDC->GetSafeHdc(), pixelformat, &pfd) == FALSE)
    {
        MessageBox("SetPixelFormat failed");
        return FALSE;
    }
    return TRUE;
}

int CModelView::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    if (CView::OnCreate(lpCreateStruct) == -1)
        return -1;

    Init();
    Init_Sphere_Data();
    SetTimer(1, 30, NULL);

    return 0;
}

void CModelView::OnDestroy()
{
    HGLRC hrc;
    hrc = ::wglGetCurrentContext();
    ::wglMakeCurrent(NULL, NULL)
    if(m_play)KillTimer(1);
    if (hrc)
        ::wglDeleteContext(hrc);
    if (m_pDC)
        delete m_pDC;
    CView::OnDestroy();
}

void CModelView::OnSize(UINT nType, int cx, int cy)
```

```
        CView::OnSize(nType, cx, cy);
        if(cy > 0)
        {
            if((m_oldRect.right > cx) || (m_oldRect.bottom > cy))
            RedrawWindow();
          m_oldRect.right = cx;
          m_oldRect.bottom = cy;
          glMatrixMode(GL_PROJECTION);
          glLoadIdentity();
            glFrustum(-1.0,1.0,-1.0,1.0,3.0,7.0);
            glViewport(0, 0, cx, cy);
      }
}

void CModelView::DrawScene()
{
        glClearColor(0.2f,0.5f,0.8f,1.0f);
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
        glColor3f(1.0f, 0.0f, 0.0f);
        glMatrixMode(GL_MODELVIEW);
        glLoadIdentity();
        glTranslatef(0.0f,0.0f,-4.5f);

        glPolygonMode(GL_FRONT_AND_BACK,m_selectMode);
        PlayBall();
        glFinish();
        SwapBuffers(wglGetCurrentDC());
}

void CModelView::OnTimer(UINT nIDEvent)
{
        Invalidate(FALSE);
        CView::OnTimer(nIDEvent);
}

void CModelView::DrawVertexArray()
{
        glDrawElements(GL_TRIANGLES, STRIANGLES*3, GL_UNSIGNED_INT, m_triSphere);
}
/*尽管采有经纬线划分法,为了提高面的精度,一个小的四边形片依然用两个小的三角形来代替,这里用经
//纬线划分所得的顶点来作为小三角形的顶点,该函数与//DrawVertexArray() 及 DrawStrips() 函数均可完
成球的绘制*/
void CModelView::DrawTriangles()
```

```
{
    int iVert, * pidx;
    glBegin(GL_TRIANGLES);
    pidx = (int *)&m_triSphere[0];
    for (iVert = 0; iVert < STRIANGLES * 3; iVert++)
    {
      Vertex( * pidx++);
    }
    glEnd();
}

void CModelView::DrawStrips()
{
    int iIdxBase;
    int iRing, iSection;

    //      绘制两极:用三角形扇,数据由经纬划分方法产生
    glBegin(GL_TRIANGLE_FAN);

    Vertex(0);
    iIdxBase = 1;
    for (iSection = 0; iSection <= SECTIONS; iSection++)
    {
      Vertex(iIdxBase + (iSection % SECTIONS));
    }
    glEnd();

    glBegin(GL_TRIANGLE_FAN);

    Vertex(SVERTICES - 1);
    iIdxBase = SVERTICES - SECTIONS - 1;
    for (iSection = SECTIONS; iSection >= 0; iSection--)
    {
      Vertex(iIdxBase + (iSection % SECTIONS));
    }
    glEnd();

    //      回归线内:用三角形片,数据由经纬划分方法产生
    iIdxBase = 1;
    for (iRing = 0; iRing < RINGS; iRing++)
    {
      glBegin(GL_TRIANGLE_STRIP);

      for (iSection = 0; iSection <= SECTIONS; iSection++)
```

```
        {
          Vertex(iIdxBase + (iSection % SECTIONS));
          Vertex(iIdxBase + (iSection % SECTIONS) + SECTIONS);
        }
        glEnd();

        iIdxBase += SECTIONS;
      }
}
//该函数通过计算获得球面上的顶点几何坐标,有限单元化小面的拓扑结构
void CModelView::GenerateSphere(float fRadius, int iRings, int iSections,
                    VERTEX *pvtx, TRIANGLE *ptri)
{
      float fTheta, fPhi;
      float fDTheta, fDPhi;
      float x, y, z, fV, fRSinTheta;
      int   i, j, n, m;
      int   iVert, iTri;

      iTri = COMPUTE_SPHERE_TRIANGLES(iRings, iSections);
      iVert = COMPUTE_SPHERE_VERTICES(iRings, iSections);

      //       获取顶部和底部顶点信息:
      pvtx[0].x = 0.0f;
      pvtx[0].y = fRadius;
      pvtx[0].z = 0.0f;
      pvtx[0].fNx = 0.0f;
      pvtx[0].fNy = 1.0f;
      pvtx[0].fNz = 0.0f;
      pvtx[0].dwColor = RGB_COLOR(0, 0, 255);
      pvtx[iVert - 1].x = 0.0f;
      pvtx[iVert - 1].y = -fRadius;
      pvtx[iVert - 1].z = 0.0f;
      pvtx[iVert - 1].fNx = 0.0f;
      pvtx[iVert - 1].fNy = -1.0f;
      pvtx[iVert - 1].fNz = 0.0f;
      pvtx[iVert - 1].dwColor = RGB_COLOR(0, 255, 0);

      //产生顶点信息:
      fDTheta = PI / (float) (iRings + 2);
      fDPhi = 2.0f * PI / (float) iSections;

      n = 1;
      fTheta = fDTheta;
```

```
//    球坐标生成法:
for (i = 0; i < = iRings; i++)
{
  y = (float)(fRadius * cos(fTheta));
  fV = fTheta / PI;
  fRSinTheta = (float)(fRadius * sin(fTheta));
  fPhi = 0.0f;

  for (j = 0; j < iSections; j++)
  {
    x = (float)(fRSinTheta * sin(fPhi));
    z = (float)(fRSinTheta * cos(fPhi));
    pvtx[n].x = x;
    pvtx[n].z = z;
    pvtx[n].y = y;
    pvtx[n].fNx = x / fRadius;
    pvtx[n].fNy = y / fRadius;
    pvtx[n].fNz = z / fRadius;
    if (n & 1)
    {
        pvtx[n].dwColor = RGB_COLOR(0, 0, 255);
    }
    else
    {
        pvtx[n].dwColor = RGB_COLOR(0, 255, 0);
    }
    fPhi + = fDPhi;
    n++;
  }
  fTheta + = fDTheta;
}

//生成三角形信息(两极):
for (i = 0; i < iSections; i++)
{
  ptri[i].iV1 = 0;
  ptri[i].iV2 = i + 1;
  ptri[i].iV3 = 1 + ((i + 1) % iSections);
  ptri[iTri - iSections + i].iV1 = iVert - 1;
  ptri[iTri - iSections + i].iV2 = iVert - 2 - i;
  ptri[iTri - iSections + i].iV3 = iVert - 2 - ((1 + i) % iSections);
}
//生成三角形信息(极回归线内)
```

```
    m = 1;
    n = iSections;

    for (i = 0; i < iRings; i++)
    {
      for (j = 0; j < iSections; j++)
      {
        ptri[n].iV1 = m + j;
        ptri[n].iV2 = m + iSections + j;
        ptri[n].iV3 = m + iSections + ((j + 1) % iSections);
        ptri[n + 1].iV1 = ptri[n].iV1;
        ptri[n + 1].iV2 = ptri[n].iV3;
        ptri[n + 1].iV3 = m + ((j + 1) % iSections);
        n + = 2;
      }
      m + = iSections;
    }
}

void CModelView::InitSphere()
{
    GenerateSphere(1.0f, RINGS, SECTIONS, m_vtxSphere, m_triSphere);

    glVertexPointer(3, GL_FLOAT, sizeof(VERTEX), &m_vtxSphere[0].x);
    glNormalPointer(GL_FLOAT, sizeof(VERTEX), &m_vtxSphere[0].fNx);
    glColorPointer(3, GL_UNSIGNED_BYTE, sizeof(VERTEX), &m_vtxSphere[0].dwColor);

    //      随机种子:(用于旋转)
    srand(time(NULL));

    m_sphere.xr = 0.0f;
    m_sphere.yr = 0.0f;
    m_sphere.zr = 0.0f;
    m_sphere.fDXr = DROT - FRANDOM(2 * DROT);
    m_sphere.fDYr = DROT - FRANDOM(2 * DROT);
    m_sphere.fDZr = DROT - FRANDOM(2 * DROT);
}

void CModelView::UpdateSphere()
{
    m_sphere.xr + = m_sphere.fDxr;
    m_sphere.yr + = m_sphere.fDyr;
    m_sphere.zr + = m_sphere.fDzr;
}
```

```
void CModelView::Vertex(int iVert)
{
    VERTEX *pvtx;
    pvtx = &m_vtxSphere[iVert];
    glColor3ubv((GLubyte *)&pvtx->dwColor);
    glNormal3fv(&pvtx->fNx);
    glVertex3fv(&pvtx->x);
}

void CModelView::Init_Sphere_Data()
{
    float fVector[4];
    int i;
    GLint dlBase;

    InitSphere();

    fVector[0] = 0.05f;
    fVector[1] = 0.05f;
    fVector[2] = 0.05f;
    fVector[3] = 1.0f;
    glLightModelfv(GL_LIGHT_MODEL_AMBIENT, fVector);

    fVector[0] = 0.0f;
    fVector[1] = 1.0f;
    fVector[2] = 1.0f;
    fVector[3] = 0.0f;
    glLightfv(GL_LIGHT0, GL_POSITION, fVector);
    fVector[0] = 0.9f;
    fVector[1] = 0.9f;
    fVector[2] = 0.9f;
    fVector[3] = 1.0f;
    glLightfv(GL_LIGHT0, GL_DIFFUSE, fVector);
    glEnable(GL_LIGHT0);
    glEnable(GL_LIGHTING);

    fVector[0] = 0.7f;
    fVector[1] = 0.7f;
    fVector[2] = 0.7f;
    fVector[3] = 1.0f;
    glMaterialfv(GL_FRONT, GL_SPECULAR, fVector);
    glMaterialf(GL_FRONT, GL_SHININESS, 50.0);
    glEnable(GL_CULL_FACE);
```

```
    //设置客户状态
    glEnableClientState(GL_VERTEX_ARRAY);
    glEnableClientState(GL_NORMAL_ARRAY);
    glEnableClientState(GL_COLOR_ARRAY);

    glColorMaterial(GL_FRONT, GL_AMBIENT_AND_DIFFUSE);
    glEnable(GL_COLOR_MATERIAL);

    //数据初始化
    dlBase = glGenLists(NLISTS);
    for (i = 0; i < NLISTS; i++)
    {
      m_iLists[i] = dlBase + i;
      glNewList(m_iLists[i], GL_COMPILE);
            if(i==0)DrawVertexArray();
            if(i==1)DrawTriangles();
            if(i==2)DrawStrips();
       glEndList();
    }
}

//完成球的绘制与旋转
void CModelView::PlayBall()
{
    DWORD dwMillis;
    glClear(GL_COLOR_BUFFER_BIT);
    glRotatef(m_sphere.xr, 1.0f, 0.0f, 0.0f);
    glRotatef(m_sphere.yr, 0.0f, 1.0f, 0.0f);
    glRotatef(m_sphere.zr, 0.0f, 0.0f, 1.0f);
    //按 m_fDisplayList 当前值选择绘制球的函数
    if (m_fDisplayList)
    {
      glCallList(m_iLists[m_iCurrentMode]);
    }
    else
    {
        if(m_iCurrentMode==0)DrawVertexArray();
        if(m_iCurrentMode==1)DrawTriangles();
        if(m_iCurrentMode==2)DrawStrips();
    }
    UpdateSphere();                                    //重设球的几
    dwMillis = GetTickCount() - dwMillis;              //时间累计
    m_dwTotalMillis += dwMillis;
    m_iTotalTriangles += STRIANGLES;                   //已绘制的三角形
```

```
}

void CModelView::ResetTotals()
{
        m_dwTotalMillis = 0;
        m_iTotalTriangles = 0;
}

void CModelView::OnKeyDown(UINT nChar, UINT nRepCnt, UINT nFlags)
{
        switch(nChar)
        {
            case VK_SPACE:          Key_Space();          break;
            case 'S':               Key_S();              break;
        }
        CView::OnKeyDown(nChar, nRepCnt, nFlags);
}

void CModelView::Key_Space()
{
        m_play = ! m_play;
        if(m_play)
            SetTimer(1,30,NULL);
        else
            KillTimer(1);
}

void CModelView::Key_S()
{
        m_controlMode = ! m_controlMode;
        if(m_controlMode)
            m_selectMode = GL_FILL;
        else
            m_selectMode = GL_LINE;
}
```

16.4 将画面输出到外部文件

你可能设计出一幅表现力和创建力非凡的画面，并需要将它作为 3D Studio Max，CorelDRAW 或 PhotoShop 等画面的背景。不可能每次到需要用的时候重新运行用 Visual C++ 生成的可执行文件，然后再进行屏幕硬拷贝，最后在画笔里面去进行剪切并保存成位图文件。有时候，你所需要的只是一部分，因而这种方法并不是一个好的方案。

本节介绍两种方法，一种方法是将画面着色到位图设备，然后再从位图设备中获取数据，第二种方法就是有选择地读取客户区像素信息，并将它保存于一个位图文件中。

16.4.1 将画面着色到位图设备

在前面所有的例程中，我们都将结果直接输出到窗口，即通过像素格式的模式旗帜为 PFD_DRAW_TO_WINDOW 来实现。要将画面输出到位图设备，可让像素格式的成员 dwFlags 支持 PFD_DRAW_TO_BITMAP 旗帜即可。本节的目标只介绍将画面着色到位图设备对象的方法。现在，请按以下步骤来完成：

(1) 建立一个新的目录

(2) 将在第 3 章中生成的 StepinGl 目录拷贝到新目录

(3)增加保护成员变量

```
int   m_nMaxX,      m_nMaxY;
HBITMAP             m_hBit;
HBITMAP             m_hOldBit;
HDC                 m_memDC;
HDC                 m_DC;
```

(4) 将如下几个函数更改如下

```
void CStepinGlView::Init()
{
    PIXELFORMATDESCRIPTOR pfd;
     int          n;
     HGLRC        hrc;
     ASSERT(m_pDC != NULL);
     if (! bSetupPixelFormat())        return;

     n = ::GetPixelFormat(m_memDC);
     ::DescribePixelFormat(m_memDC, n, sizeof(pfd), &
     hrc = wglCreateContext(m_memDC);
     wglMakeCurrent(m_memDC, hrc);

     glClearDepth(1.0f);
     glEnable(GL_DEPTH_TEST);
     glMatrixMode(GL_MODELVIEW);
     glLoadIdentity();
}

BOOL CStepinGlView::bSetupPixelFormat()
{
     static PIXELFORMATDESCRIPTOR pfd =
     {
```

```
        sizeof(PIXELFORMATDESCRIPTOR),
        1,
        PFD_DRAW_TO_WINDOW |
          PFD_SUPPORT_OPENGL |
            PFD_DRAW_TO_BITMAP,
        PFD_TYPE_RGBA,
        24,
        0, 0, 0, 0, 0, 0,
        0,
        0,
        0,
        0, 0, 0, 0,
        32,
        0,
        0,
        PFD_MAIN_PLANE,
        0,
        0, 0, 0
    };
    int pixelformat;
    if ( (pixelformat = ChoosePixelFormat(m_memDC, &pfd)) == 0 )
    {
      MessageBox("ChoosePixelFormat failed");
      return FALSE;
    }
    if (SetPixelFormat(m_memDC, pixelformat, &pfd) == FALSE)
    {
      MessageBox("SetPixelFormat failed");
      return FALSE;
    }
    return TRUE;
}

int CStepinGlView::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    if (CView::OnCreate(lpCreateStruct) == -1)
        return -1;

    // TODO: Add your specialized creation code here

    m_pDC = new CClientDC(this);

    GetClientRect(&m_oldRect);
```

```
    //获得屏幕大小:
    m_nMaxx = GetSystemMetrics(SM_CXSCREEN);
    m_nMaxy = GetSystemMetrics(SM_CYSCREEN);

    //创建显示内存设备并关联:
    m_DC = m_pDC->GetSafeHdc();
    m_memDC = ::CreateCompatibleDC(m_DC);
    m_hBit = CreateCompatibleBitmap(m_DC,300,300);

    if(m_hBit==NULL)MessageBox("Create DIB failed");
    m_hOldBit = (HBITMAP)SelectObject(m_memDC,m_hBit);

    Init();
    return 0;
}

void CStepinGlView::OnSize(UINT nType, int cx, int cy)
{
    CView::OnSize(nType, cx, cy);
    // TODO: Add your message handler code here
    if(cy > 0)
    {
        if((m_oldRect.right > cx) || (m_oldRect.bottom > cy))
        RedrawWindow();
      m_oldRect.right = cx;
      m_oldRect.bottom = cy;
      glMatrixMode(GL_PROJECTION);
      glLoadIdentity();
        glFrustum(-1.0,1.0,-1.0,1.0,3.0,7.0);
        glViewport(0, 0, cx, cy);
    }
}

void CStepinGlView::OnDestroy()
{
    // TODO: Add your message handler code here
    HGLRC hrc;
    hrc = ::wglGetCurrentContext();
    ::wglMakeCurrent(NULL, NULL);
    if (hrc)
      ::wglDeleteContext(hrc);
    if(m_hOldBit)::DeleteObject(m_hOldBit);
    if(m_hBit)::DeleteObject(m_hBit);
    if(m_memDC)::DeleteDC(m_memDC);
```

```
    if(m_DC)::DeleteDC(m_DC);
    if (m_pDC)
      delete m_pDC;
    CView::OnDestroy();
}

void CStepinGlView::OnDraw(CDC* pDC)
{
    CStepinGlDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);

    DrawScene();

    //    下面是在函数中的处理:(从内存读入数据)
    CRect rectClip;
    pDC->GetClipBox(&rectClip);
    GLint viewport[4];
    glGetIntegerv (GL_VIEWPORT, viewport);

    //read in:
    ::BitBlt(pDC->m_hDC,m_oldRect.left,m_oldRect.top,
        viewport[2],viewport[3],
        m_memDC,0,0,
        SRCCOPY);
}
```

下面这个函数在文件 MainFrm.cpp 中:

```
BOOL CMainFrame::PreCreateWindow(CREATESTRUCT& cs)
{
    cs.cx=300;
    cs.cy=335;

    return CFrameWnd::PreCreateWindow(cs);
}
```

到此,已经可以在位图设备对象中着色画面了。

(5) 添加保护成员函数:DrawSphere()

```
void CStepinGlView::DrawSphere()
{
    GLUquadricObj* quadObj;
    quadObj=gluNewQuadric();
    gluQuadricDrawStyle(quadObj,GLU_FILL);
    gluQuadricNormals(quadObj,GLU_SMOOTH);
    gluSphere(quadObj,1.0,32,32);
```

```
}
```

(6) 编辑 DrawScene()函数

```
void CStepinGlView::DrawScene()
{
    glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glTranslatef(0.0f,0.0f, -4.5f);
    glColor3f(1.0f, 0.0f, 0.0f);
    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    DrawSphere();
    glFinish();
}
```

(7)运行程序

编译连接并运行程序,画面闪烁一下之后出现一个球。当然现在还没有把画面保存在外部文件中,紧接着需要完成的任务就是读取位图设备中的数据信息。请读者自己编程。

16.4.2 有选择地输出画面

有时候,我们只需要画面的一个部分,这就要对画面进行有选择性地输出。要输出一部分画面的方法很多,比如,将画面画到内存位图设备对象,然后再选择一个子图进行输出,这种处理方法通用易行,依赖于 Windows 处理位图的函数。在输出时,颜色信息有可能遭到损失。下面介绍的方法只依赖 OpenGL 的 glReadPixels()函数。

在制作的过程中,读者将能学习到:

第一,如何有选择性地将画面输出到位图;

第二,画面的艺术处理效果;

第三,光线的组合效果;

第四,程序的组织。

请按如下步骤完成编程:

(1) 建立工程

建立一个名为 ToDib 的工程,并按第 5 章中的方法极小化 OpenGL。最后请参考一下程序清单,看看是否有不同之处。

(2) 添加一个对话框

为工程添加一个对话框,如图 16-6 所示,并按表 16-2 编辑对话框。

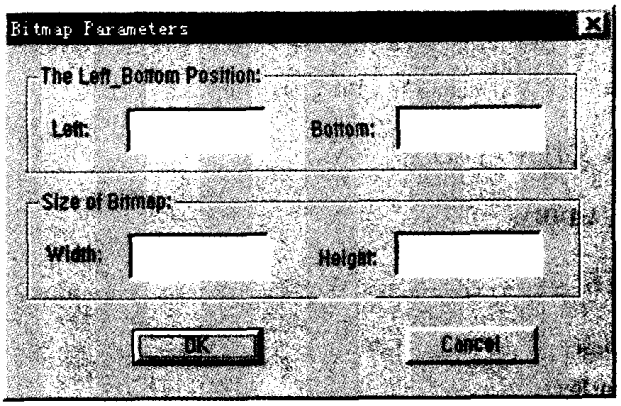

图 16-6　对话框

表 16-2

控 件	ID	Caption
对话框	IDD_DIALOG1	Bitmap Parameters
组	IDC_STATIC	The Left_Bottom Position:
组	IDC_STATIC	Size of Bitmap:
静态文本框	IDC_STATIC	Left:
静态文本框	IDC_STATIC	Bottom:
静态文本框	IDC_STATIC	Width:
静态文本框	IDC_STATIC	Height:
编辑框	IDC_LEFTX	
编辑框	IDC_BOTTONY	
编辑框	IDC_WIDTH	
编辑框	IDC_HEIGHT	

为该对话框增加一个类,类名为 CToDibDlg。按表 16-3 添加成员变量。

表 16-3

控　件	成　员	类　型
IDC_LEFTX	m_leftX	int
IDC_BOTTONY	m_bottomY	int
IDC_WIDTH	m_width	int
IDC_HEIGHT	m_height	int

(3) 添加菜单并响应菜单

添加一个 Caption 名为 &To_Dib 的菜单。加入一个菜单项。其 Caption 为 S&ave as Dib,其 ID 号为 IDM_SAVE_TO_DIB。再添加一个 Caption 名为 S&et 的菜单。加入一个菜单项。其 Caption 为 &Bitmap Parameters ...,其 ID 号为 IDM_DIB_SET。并分别响应菜单。注意,在编辑 OnSaveToDib()函数时,请注意如下语句(见程序清单)应对应于你的确定目

录：

```
CString    FileName = "e: \ \ xsmimag \ \ DibTest1.bmp";
```

(4) 在 ToDibView.h 文件中增加句柄申明

```
DECLARE_HANDLE(HDIB);
```

(5) 添加保护成员变量

```
GLint      m_iListBase;
CRect      m_currentRect;
int        m_dibWidth;
int        m_dibHeight;
int        m_dibLeft;
int        m_dibBottom;
```

(6) 添加保护成员函数，并按构造函数初始化

```
void       DrawSphere();
void       DrawBack();
void       InitObjects();
void       CalcuNormal(GLfloat * fV,GLfloat * fN);
void       Circle(GLfloat dRadius,GLfloat dThick,GLint n);
void       EnableLighting();
void       Artistic();
```

请按程序列表完成诸函数的编辑。

(7) 添加宏定义和增加文件包含

在 ToDibView.cpp 文件的头部添加宏定义和文件包含：

```
# define    PI    3.14159265358979323846
# include          "math.h"
# include          "ToDibDlg.h"
```

(8) 编辑 DrawScene()函数

(9) 运行程序

编译连接并运行程序。程序执行后会出现如图 16-7 所示的画面。执行菜单 To_Dib \ Save as Dib，结果将生成一个位图。可以在画笔中打开该位图，当然也可将此位图输入到其他图形图像制作软件中去。执行菜单 Set \ Bitmap Parameters ...，结果出现一个对话框，在对话框中可以输入位图的左下角在客户区中的位置(窗口坐标)、位图的宽度和高度。最后，应用程序将对位图的宽度和高度进行处理，使其能被 4 整除。

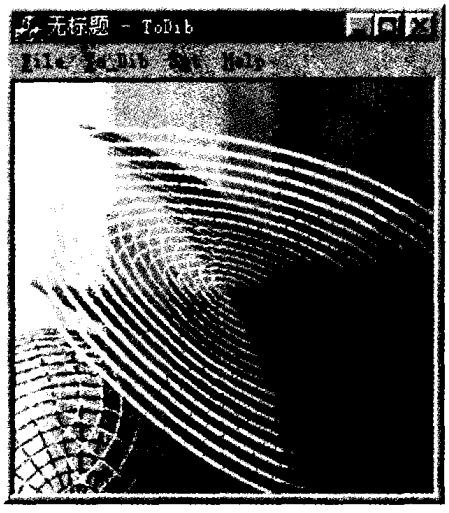

图 16-7　输出画面

16.4.3　程序清单

ToDibView.h 文件：

```
//////////////////////////////////////////////////////////////////////////////////////////////
//////////////////////////                                      //////////////////////////
//////////////////////////            ToDibView.h               //////////////////////////
//////////////////////////                                      //////////////////////////
//////////////////////////////////////////////////////////////////////////////////////////////
// ToDibView.h : interface of the CToDibView class
#include "gl\gl.h"
#include "gl\glu.h"

#if !defined(AFX_TODIBVIEW_H__7C3B2DEF_805E_11D2_83B4_0080C832FCF3__INCLUDED_)
#define AFX_TODIBVIEW_H__7C3B2DEF_805E_11D2_83B4_0080C832FCF3__INCLUDED_

#if _MSC_VER >= 1000
#pragma once
#endif // _MSC_VER >= 1000

DECLARE_HANDLE(HDIB);                    //申明一个内存对象句柄

class CToDibView : public CView
{
protected: // create from serialization only
    CToDibView();
    DECLARE_DYNCREATE(CToDibView)
```

```
// Attributes
public:
        CToDibDoc * GetDocument();

        CClientDC      * m_pDC;
        CRect          m_oldRect;

// Operations
public:

// Overrides
        // ClassWizard generated virtual function overrides
        //{{AFX_VIRTUAL(CToDibView)
        public:
        virtual void OnDraw(CDC * pDC); // overridden to draw this view
        virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
        protected:
        //}}AFX_VIRTUAL

// Implementation
public:
        virtual ~CToDibView();
#ifdef _DEBUG
        virtual void AssertValid() const;
        virtual void Dump(CDumpContext& dc) const;
#endif

protected:
        GLint     m_iListBase;                  //显示列表基数
        CRect     m_currentRect;                //当前窗口大小
        int       m_dibWidth;                   //位图的宽度
        int       m_dibHeight;                  //位图的高度
        int       m_dibLeft;                    //位图在客户区中的左下角点的 x 坐标
        int       m_dibBottom;                  //位图在客户区中的左下角点的 y 坐标

// Generated message map functions
protected:
        void DrawSphere();
        void DrawBack();
        void InitObjects();
        void CalcuNormal(GLfloat * fV,GLfloat * fN);
        void Circle(GLfloat dRadius, GLfloat dThick,GLint n);
        void EnableLighting();
        void Artistic();
```

```
        void DrawScene();
        BOOL bSetupPixelFormat();
        void Init();
        //{{AFX_MSG(CToDibView)
        afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
        afx_msg void OnDestroy();
        afx_msg void OnSize(UINT nType, int cx, int cy);
        afx_msg void OnSaveToDib();
        afx_msg void OnDibSet();
        //}}AFX_MSG
        DECLARE_MESSAGE_MAP()
};

#ifndef _DEBUG  // debug version in ToDibView.cpp
inline CToDibDoc* CToDibView::GetDocument()
        { return (CToDibDoc*)m_pDocument; }
#endif

/////////////////////////////////////////////////////////////////////////////

//{{AFX_INSERT_LOCATION}}
// Microsoft Developer Studio will insert additional declarations immediately before the previous line.

#endif // !defined(AFX_TODIBVIEW_H__7C3B2DEF_805E_11D2_83B4_0080C832FCF3__
                INCLUDED_)
```

ToDibView.cpp 文件：

```
/////////////////////////////////////////////////////////////////////////////
//////////////////////////                                 //////////////////////////
//////////////////////////          ToDibView.cpp          //////////////////////////
//////////////////////////                                 //////////////////////////
/////////////////////////////////////////////////////////////////////////////

#include "stdafx.h"
#include "ToDib.h"

#include "ToDibDoc.h"
#include "ToDibView.h"

#include "math.h"
#include "ToDibDlg.h"
```

```
#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

#define PI    3.14159265358979323846

IMPLEMENT_DYNCREATE(CToDibView, CView)

BEGIN_MESSAGE_MAP(CToDibView, CView)
    //{{AFX_MSG_MAP(CToDibView)
    ON_WM_CREATE()
    ON_WM_DESTROY()
    ON_WM_SIZE()
    ON_COMMAND(IDM_SAVE_TO_DIB, OnSaveToDib)
    ON_COMMAND(IDM_DIB_SET, OnDibSet)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CToDibView construction/destruction

CToDibView::CToDibView()
{
    m_pDC = NULL;

    m_dibWidth = 64;
    m_dibHeight = 64;
    m_dibLeft = 0;
    m_dibBottom = 0;
}

CToDibView::~CToDibView()
{
}

BOOL CToDibView::PreCreateWindow(CREATESTRUCT& cs)
{
    cs.style |= WS_CLIPCHILDREN | WS_CLIPSIBLINGS;
    return CView::PreCreateWindow(cs);
}

/////////////////////////////////////////////////////////////////////////////
```

```
// CToDibView drawing

void CToDibView::OnDraw(CDC* pDC)
{
	CToDibDoc* pDoc = GetDocument();
	ASSERT_VALID(pDoc);

	DrawScene();
}

/////////////////////////////////////////////////////////////////////////////
// CToDibView diagnostics

#ifdef _DEBUG
void CToDibView::AssertValid() const
{
	CView::AssertValid();
}

void CToDibView::Dump(CDumpContext& dc) const
{
	CView::Dump(dc);
}

CToDibDoc* CToDibView::GetDocument() // non-debug version is inline
{
	ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(CToDibDoc)));
	return (CToDibDoc*)m_pDocument;
}
#endif //_DEBUG

/////////////////////////////////////////////////////////////////////////////
// CToDibView message handlers

void CToDibView::Init()
{
	PIXELFORMATDESCRIPTOR pfd;
	int         n;
	HGLRC       hrc;
	m_pDC = new CClientDC(this);
	ASSERT(m_pDC != NULL);
	if (!bSetupPixelFormat())
	  return;
```

```
    n = ::GetPixelFormat(m_pDC->GetSafeHdc());
    ::DescribePixelFormat(m_pDC->GetSafeHdc(), n, sizeof(pfd), &pfd);
    hrc = wglCreateContext(m_pDC->GetSafeHdc());
    wglMakeCurrent(m_pDC->GetSafeHdc(), hrc);

    GetClientRect(&m_oldRect);
    glClearDepth(1.0f);
    glEnable(GL_DEPTH_TEST);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

BOOL CToDibView::bSetupPixelFormat()
{
    static PIXELFORMATDESCRIPTOR pfd =
    {
      sizeof(PIXELFORMATDESCRIPTOR),
      1,
      PFD_DRAW_TO_WINDOW |
        PFD_SUPPORT_OPENGL,
      PFD_TYPE_RGBA,
      24,
      0, 0, 0, 0, 0, 0,
      0,
      0,
      0,
      0, 0, 0, 0,
      32,
      0,
      0,
      PFD_MAIN_PLANE,
      0,
      0, 0, 0
    };
    int pixelformat;
    if ( (pixelformat = ChoosePixelFormat(m_pDC->GetSafeHdc(), &pfd)) == 0 )
    {
      MessageBox("ChoosePixelFormat failed");
      return FALSE;
    }
    if (SetPixelFormat(m_pDC->GetSafeHdc(), pixelformat, &pfd) == FALSE)
    {
      MessageBox("SetPixelFormat failed");
      return FALSE;
```

```
    }
    return TRUE;
}

int CToDibView::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    if (CView::OnCreate(lpCreateStruct) == -1)
        return -1;

    Init();
    InitObjects();

    return 0;
}

void CToDibView::OnDestroy()
{
    HGLRC hrc;
    hrc = ::wglGetCurrentContext();
    ::wglMakeCurrent(NULL, NULL);
    if (hrc)
      ::wglDeleteContext(hrc);
    if (m_pDC)
      delete m_pDC;
    CView::OnDestroy();
}

void CToDibView::OnSize(UINT nType, int cx, int cy)
{
    CView::OnSize(nType, cx, cy);
    if(cy > 0)
    {
        if((m_oldRect.right > cx) || (m_oldRect.bottom > cy))
        RedrawWindow();
      m_oldRect.right = cx;
      m_oldRect.bottom = cy;
      glMatrixMode(GL_PROJECTION);
      glLoadIdentity();
        glFrustum(-1.0,1.0,-1.0,1.0,3.0,15.0);
        glViewport(0, 0, cx, cy);
    }
}

void CToDibView::DrawScene()
```

```
    glClearColor(1.0f,1.0f,1.0f,0.5f);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glShadeModel(GL_SMOOTH);

    DrawBack();
    glTranslatef(0.0f,0.0f, -8.0f);
    EnableLighting();
    Artistic();

    glFinish();
    glDisable(GL_LIGHTING);
}

void CToDibView::Artistic()
{
    glEnable(GL_LINE_SMOOTH);
    glEnable(GL_BLEND);
    glBlendFunc(GL_SRC_ALPHA,GL_ONE_MINUS_SRC_ALPHA);
    glHint(GL_POLYGON_SMOOTH_HINT,GL_NICEST);
    //调用显示列表序列:绘制圆圈
    GLbyte BaseArray[3] = {0,1,2};
    glCallLists(3,GL_BYTE,BaseArray);

    glDisable(GL_LINE_SMOOTH);
    glDisable(GL_BLEND);
}

void CToDibView::EnableLighting()
{
    //创一般光源
    GLfloat light0Amb[] = {0.0f,0.0f,0.0f,1.0f};
    GLfloat light0Dif[] = {0.9f,0.9f,0.9f,1.0f};
    GLfloat light0Pos[] = {0.0f,10.0f,15.0f,1.0f};
    glLightfv(GL_LIGHT0,GL_AMBIENT,light0Amb);
    glLightfv(GL_LIGHT0,GL_DIFFUSE,light0Dif);
    glLightfv(GL_LIGHT0,GL_POSITION,light0Pos);
    //创建聚光
    GLfloat spotAmb[] = {0.2f,0.2f,0.2f,0.8f};
    GLfloat spotDif[] = {1.0f,1.0f,1.0f,0.8f};
    GLfloat spotSpe[] = {0.9f,0.9f,0.7f,0.8f};
    GLfloat spotPos[] = { -10.0f,1.0f,10.0f,0.8f};
```

```
    GLfloat spotDir[] = {1.0f,0.0f, -1.0f,1.0f};
    GLfloat spotCutoff = 38.0f;
    glLightfv(GL_LIGHT1,GL_AMBIENT,spotAmb);
    glLightfv(GL_LIGHT1,GL_DIFFUSE,spotDif);
    glLightfv(GL_LIGHT1,GL_SPECULAR,spotSpe);
    glLightfv(GL_LIGHT1,GL_POSITION,spotPos);
    glLightfv(GL_LIGHT1,GL_SPOT_DIRECTION,spotDir);
    glLightf(GL_LIGHT1,GL_SPOT_CUTOFF,spotCutoff);
    glLightf(GL_LIGHT1,GL_CONSTANT_ATTENUATION,0.3);

    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    glEnable(GL_LIGHT1);
}
void CToDibView::Circle(GLfloat dRadius, GLfloat dThick,GLint n)
{
    GLfloat dStep = (2 * PI)/(GLfloat)n;
    GLfloat fV[3],fN[3];
    glLineWidth(dThick);
    //创建圆圈
    glBegin(GL_LINE_STRIP);
        for(GLdouble i = 0.0;i <= 2 * PI + dStep;i += dStep)
        {
            fV[0] = dRadius * cos(i);
            fV[1] = dRadius * sin(i);
            fV[2] = 0.0f;
            CalcuNormal(fV,fN);
            glNormal3fv(fN);
            glVertex3fv(fV);
        }
    glEnd();
}

void CToDibView::CalcuNormal(GLfloat * fV,GLfloat * fN)
{
    GLfloat fL = (fV[0] * fV[0] + fV[1] * fV[1] + fV[2] * fV[2]);
    fL = sqrt(fL);
    if(fL < 0.00001)fL = 1.0f;
    //法线的归一化
    fN[0] = fV[0]/fL;
    fN[1] = fV[1]/fL;
    fN[2] = fV[2]/fL;
}
```

```
void CToDibView::InitObjects()
{
    GLfloat mat_amb[4] = {0.2f, 0.2f, 0.05f,0.8f};
    GLfloat mat_dif[4] = {1.0f,0.6f,0.1f,0.8f};
    GLfloat mat_spe[4] = {0.8f,0.8f,0.8f,0.8f};
    GLfloat mat_emi[4] = {0.2f,0.2f,0.2f,0.8};
    GLfloat shininess = 60;

    m_iListBase = glGenLists(3);
    glListBase(m_iListBase);
    //用显示列表管理材质
    glNewList(m_iListBase,GL_COMPILE);
        glMaterialfv (GL_FRONT, GL_AMBIENT, mat_amb);
        glMaterialfv (GL_FRONT, GL_DIFFUSE, mat_dif);
        glMaterialfv (GL_FRONT, GL_SPECULAR, mat_spe);
        glMaterialfv(GL_FRONT,GL_EMISSION,mat_emi);
        glMaterialf (GL_FRONT, GL_SHININESS,shininess );

        glPushMatrix();
            glRotatef(-105.0f,1.0f,0.0f,1.0f);
            GLfloat lw[2];
            glGetFloatv(GL_LINE_WIDTH_RANGE,lw);
            //位于同一平面上的不同半径的圆圈
            Circle(0.2,lw[1]/8.0,50);
            Circle(0.4,lw[1]/8.0,50);
            Circle(0.6,lw[1]/8.0,50);
            Circle(0.8,lw[1]/8.0,100);
            Circle(1.0,lw[1]/7.0,100);
            Circle(1.2,lw[1]/7.0,100);
            Circle(1.4,lw[1]/7.0,100);
            Circle(1.6,lw[1]/7.0,100);
            Circle(1.8,lw[1]/6.0,100);
            Circle(2.0,lw[1]/6.0,100);
            Circle(2.2,lw[1]/6.0,100);
            Circle(2.4,lw[1]/6.0,100);
            Circle(2.6,lw[1]/4.0,100);
            Circle(2.8,lw[1]/4.0,100);
            Circle(3.0,lw[1]/4.0,100);
            Circle(3.2,lw[1]/4.0,100);
        glPopMatrix();
    glEndList();

    glNewList(m_iListBase+1,GL_COMPILE);
        glPushMatrix();
```

```
            glRotatef(210.0f,0.0f,0.0f,1.0f);
            glCallList(m_iListBase);
        glPopMatrix();
    glEndList();
    DrawSphere();
}

void CToDibView::DrawBack()
{
    glFrontFace(GL_CW);
    //一个光顺着色处理的背景
    glBegin(GL_QUADS);
        glColor4f(1.0f,1.0f,0.0f,0.5f);
        glVertex3f(-5.0f,5.0f,-14.0f);
        glColor4f(0.0f,1.0f,1.0f,0.5f);
        glVertex3f(-5.0f,-5.0f,-14.0f);
        glColor4f(1.0f,0.0f,0.0f,0.5f);
        glVertex3f(5.0f,-5.0f,-14.0f);
        glColor4f(1.0f,0.0f,1.0f,0.5f);
        glVertex3f(5.0f,5.0f,-14.0f);
    glEnd();
}

void CToDibView::DrawSphere()
{
    GLfloat mat_amb[4] = {0.2f, 0.2f, 0.05f,0.8f};
    GLfloat mat_dif[4] = {0.70f,0.7f,0.20f,0.8f};
    GLfloat mat_spe[4] = {0.8f,0.8f,0.8f,0.8f};
    GLfloat mat_emi[4] = {0.2f,0.0f,0.1f,0.8};
    GLfloat shininess = 50;
    GLfloat lw[2];
    glGetFloatv(GL_LINE_WIDTH_RANGE,lw);

    GLUquadricObj* quadObj;
    glNewList(m_iListBase+2,GL_COMPILE);
        glLineWidth(lw[1]/8.0);
        glMaterialfv(GL_FRONT, GL_AMBIENT, mat_amb);
        glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_dif);
        glMaterialfv(GL_FRONT, GL_SPECULAR, mat_spe);
        glMaterialfv(GL_FRONT,GL_EMISSION,mat_emi);
        glMaterialf(GL_FRONT, GL_SHININESS,shininess);

        glPushMatrix();
            glTranslatef(-2.5f, -2.6f, -1.0f);
```

```
            glRotatef(140.0f,1.0f,0.0f,0.0f);
            quadObj = gluNewQuadric();
            gluQuadricDrawStyle(quadObj,GLU _ LINE);
            gluQuadricNormals(quadObj,GLU _ FLAT);
            gluSphere(quadObj,2.0,24,24);
        glPopMatrix();
    glEndList();
}

void CToDibView::OnSaveToDib()
{
    BeginWaitCursor();

    BITMAPFILEHEADER bmfHdr;                    //位图文件头信息
    LPBITMAPINFOHEADER lpBI;                    //位图信息
    DWORD dwBitsSize;                           //位图位数据的大小
    HDIB hDIB;                                  //内存句柄
    LPSTR pDIBits;                              //位图位数据指针

    dwBitsSize = sizeof(BITMAPINFOHEADER) + m _ dibWidth * m _ dibHeight * 3;
    //内存分配
    hDIB = (HDIB)::GlobalAlloc(GMEM _ MOVEABLE GMEM _ ZEROINIT, dwBitsSize + 32);
    if (hDIB = = 0)     return ;
    //内存指定与指针计算
    lpBI = (LPBITMAPINFOHEADER)::GlobalLock((HGLOBAL) hDIB);
    if (lpBI = = NULL) return;

    pDIBits = (LPSTR)lpBI + sizeof(BITMAPINFOHEADER);

    //填充文件及位图信息结构
    bmfHdr.bfType = 0x4d42;
    bmfHdr.bfSize = sizeof(BITMAPFILEHEADER) + sizeof(BITMAPINFOHEADER)
                    + m _ dibWidth * m _ dibHeight * 3;
    bmfHdr.bfReserved1 = 0;
    bmfHdr.bfReserved2 = 0;
    bmfHdr.bfOffBits = sizeof(BITMAPFILEHEADER) + sizeof(BITMAPINFOHEADER);

    lpBI - > biSize = sizeof(BITMAPINFOHEADER);
    lpBI - > biWidth = m _ dibWidth;
    lpBI - > biHeight = m _ dibHeight;
    lpBI - > biPlanes = 1;
    lpBI - > biBitCount = 24;
    lpBI - > biCompression = 0;
    lpBI - > biSizeImage = 0;
```

```
        lpBI->biXPelsPerMeter = 0;
        lpBI->biYPelsPerMeter = 0;
        lpBI->biClrUsed = 0;
        lpBI->biClrImportant = 0;
        //获取客户区(视区)像素数据
        glReadPixels(m_dibLeft,m_dibBottom, m_dibWidth,m_dibHeight,
            GL_BGR_EXT,GL_UNSIGNED_BYTE,(GLubyte*)pDIBits);

        //请创建自己的目录或获取当前目录:
        CString FileName="e:\\xsmimag\\DibTest1.bmp";
        //创建文件
        CFile file(FileName,CFile::modeCreate |
            CFile::modeReadWrite | CFile::shareExclusive);
        //写入数据
        file.Write((LPSTR)&bmfHdr,sizeof(BITMAPFILEHEADER));
        file.WriteHuge((LPSTR)lpBI,dwBitsSize);
        file.Close;

        ::GlobalUnlock((HGLOBAL) hDIB);

        //在此处,现在可以无效化句柄 hDIB:
        ::GlobalFree((HGLOBAL) hDIB);

        EndWaitCursor();
}

void CToDibView::OnDibSet()
{
        CToDibDlg dlg;
        GetClientRect(&m_currentRect);
        //用对话框设置位图大小与数据位置
        dlg.m_width = m_dibWidth;
        dlg.m_height = m_dibHeight;
        dlg.m_leftX = m_dibLeft;
        dlg.m_bottomY = m_dibBottom;

        int responeDlg=dlg.DoModal();
        if(responeDlg == IDOK)
        {
            m_dibWidth=dlg.m_width;
            m_dibHeight=dlg.m_height;
            m_dibLeft=dlg.m_leftX;
            m_dibBottom=dlg.m_bottomY;
        }
```

```
    //数据的优化处理
    if(m_dibWidth > 4*((m_currentRect.Width()/4) - 1) - m_dibLeft)
    {
        m_dibWidth = 4*(((m_currentRect.Width() - m_dibLeft)/4) - 1);
    }
    if(m_dibHeight > 4*((m_currentRect.Height()/4) - 1) - m_dibBottom)
    {
        m_dibHeight = 4*(((m_currentRect.Height() - m_dibBottom)/4) - 1);
    }
```

ToDibDlg.h 文件：

```
////////////////////////////////////////////////////////////////////////////////////////////
//////////////////////////                                  //////////////////////////
//////////////////////////          ToDibDlg.h              //////////////////////////
//////////////////////////                                  //////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////
#if !defined(AFX_TODIBDLG_H__217BE4A3_80B1_11D2_83B4_0080C832FCF3__INCLUDED_)
#define AFX_TODIBDLG_H__217BE4A3_80B1_11D2_83B4_0080C832FCF3__INCLUDED_

#if _MSC_VER >= 1000
#pragma once
#endif // _MSC_VER >= 1000
// ToDibDlg.h : header file
//
/////////////////////////////////////////////////////////////////////////////
// CToDibDlg dialog

class CToDibDlg : public CDialog
{
// Construction
public:
    CToDibDlg(CWnd* pParent = NULL);    // standard constructor

// Dialog Data
    //{{AFX_DATA(CToDibDlg)
    enum { IDD = IDD_DIALOG1 };
    int         m_leftX;
    int         m_height;
    int         m_width;
    int         m_bottomY;
    //}}AFX_DATA

// Overrides
```

```
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CToDibDlg)
    protected:
    virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV support
    //}}AFX_VIRTUAL

// Implementation
protected:

    // Generated message map functions
    //{{AFX_MSG(CToDibDlg)
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};

//{{AFX_INSERT_LOCATION}}
// Microsoft Developer Studio will insert additional declarations immediately before the previous line.

#endif // !defined(AFX_TODIBDLG_H__217BE4A3_80B1_11D2_83B4_0080C832FCF3__INCLUDED_)
```

ToDibDlg.cpp 文件：

```
////////////////////////////////////////////////////////////////////////////////////////////////
//////////////////////////                                        //////////////////////////
//////////////////////////            ToDibDlg.cpp                //////////////////////////
//////////////////////////                                        //////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////////

// ToDibDlg.cpp : implementation file

#include "stdafx.h"
#include "ToDib.h"
#include "ToDibDlg.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
// CToDibDlg dialog

CToDibDlg::CToDibDlg(CWnd* pParent /* = NULL */)
```

```
    : CDialog(CToDibDlg::IDD, pParent)
{
    //{{AFX_DATA_INIT(CToDibDlg)
    m_leftX = 0;
    m_height = 0;
    m_width = 0;
    m_bottomY = 0;
    //}}AFX_DATA_INIT
}

void CToDibDlg::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    //{{AFX_DATA_MAP(CToDibDlg)
    DDX_Text(pDX, IDC_LEFTX, m_leftX);
    DDX_Text(pDX, IDC_HEIGHT, m_height);
    DDX_Text(pDX, IDC_WIDTH, m_width);
    DDX_Text(pDX, IDC_BOTTONY, m_bottomY);
    //}}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(CToDibDlg, CDialog)
    //{{AFX_MSG_MAP(CToDibDlg)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CToDibDlg message handlers
```

16.5 制作阴影

事实上,我们已经介绍了很多真实化场景的方法,但是那些方法都忽略了一个问题:当光照射物体时,不会产生阴影。这与自然现象是不相符的。因此,我们有必要来研究物体阴影的制作。遗憾的是,制作阴影和生成阴影决不是一件简单的事。本节将介绍制作阴影的一般方法。

16.5.1 解决问题的途径

物体产生阴影是因为该物体将光线挡住了,怎样用程序实现这一点呢?我们没有必要说:"这是 OpenGL 的缺陷"。一般地,可以采用数学计算的方法来实现。分两步完成:

第一步,要解决如下几个问题:

第一,物体的几何拓扑关系要简洁明确。需要管理物体的几何顶点,明确单面是由哪些顶点构成的,以及这些顶点的排列顺序,面的法线等几何要素和几何拓扑关系。

第二,怎样通过光的透视来获得一个几何顶点的平面透视点。需要考虑如下问题:给定光源,怎样求其透视点;给定光的照射方向,怎样求一个点的透视点等等。

第三,怎样获得光的反射方向。

第四,怎样确定背光的面。背光的面一般不会产生阴影。

第五,当确定了物体几何点的透视点后,怎样来绘制阴影。

第六,当阴影比较钝时,如何绘制半影。

第七,当阴影被物体挡住时,怎样在其他物体上绘制阴影。

当然,绘制阴影所面临的技术问题绝不只是这些。本节将解决前面6个问题。

第二步,要考虑算法的实现:

第一,用空间解析几何的知识(或欧氏几何、仿射几何、射影几何与微分几何)解决点(空间的一个点就是一个向量)的平移、旋转和缩放等变换,解决点的透视、光的反射等问题。可以建立一个专门处理点的类,该类应该处理如下问题:向量的加减运算、向量与标量的数乘运算、向量与向量的点积运算 、向量与向量的向量积运算、向量的单位化、向量的范数、向量的平移、向量的缩放、点绕 xyz 轴的旋转、点绕任一给定轴的旋转、点的透视点、向量(代表光方向)的反射。

第二,建立物体。物体要具有通用性,并具有良好的可交互性。可通过创建类的方式来创建物体。比如,铅笔是一类物体,铅笔一般由笔尖、笔杆和笔帽等部分构成,这是其共性,其个性是铅笔的颜色是可以进行选择的,铅笔的位置或姿态可能不同。在创建铅笔类时,其共性可以通过几何建模来完成,其个性可通过参数由人机交互方式来完成。该类不仅能绘制几何体,而且可以绘制其阴影。具体地说,该类应该完成以下工作:对几何体的顶点进行管理,应能在外部通过参数自动控制其顶点;能在外部控制物体在空间的位置,当位置发生变化时,光照应自动进行纠正;能确定背光的面;能在外部控制物体的材质,能控制阴影的颜色;能绘制几何体;能绘制阴影;能绘制有半影的阴影等等。

本节的任务是要绘制一支铅笔,并能将铅笔的阴影绘制在 o_xy 平面上。

16.5.2 绘制铅笔及其阴影

为了简化问题,此处代表一类物体的"铅笔"由一个笔帽、笔杆和笔尖构成。其中,笔杆为一个六面体,笔尖为一个六面锥体,笔帽为一个简单的六边形。现在来完成绘制工作。

(1) 建立工程

建立一个名为 Shadow 的工程,并按第5章中的方法极小化 OpenGL。最后参考一下程序清单,看看是否有不同之处。这里需要重新编辑 OnSize()函数。

(2) 建立一个名为 Point 的类,该类处理点或向量的各种运算和透视等问题。请按程序清单进行编辑。在编辑时,注意类的层次关系,各变量、函数的类型,文件包含等。

(3) 建立一个名为 Color 的类,该类处理颜色的赋值和与颜色有关的操作。请按程序清单进行编辑。在编辑时,注意类的层次关系,各变量、函数的类型,文件包含等。

(4) 建立一个名为 Pencil 的类,该类负责铅笔的绘制、铅笔阴影的绘制、人机交互等工作。请按程序清单进行编辑。在编辑时,注意类的层次关系,各变量、函数的类型,文件包含等。

(5) 为 CShadowView 类添加保护成员变量

```
GLfloat        m_lightPos_x;
GLfloat        m_lightPos_y;
GLfloat        m_lightPos_z;
```

注意对其进行如下初始化:

```
m_lightPos_x = 1.0f;
m_lightPos_y = -1.0f;
m_lightPos_z = 1.3f;
```

(6) 为 CShadowView 类添加保护成员函数

```
void        Ground();
void        CreateSun();
```

请按程序清单完成有关它们的编辑。

(7) 编辑函数 DrawScene()

请按程序清单自己完成编辑。

(8) 运行程序

编译连接并运行程序,画面上出现了一只红色的铅笔,其笔芯为红木所制,如图 16-8 所示。

请注意,可以将铅笔进行平移、缩放和绕 x、y、z 轴进行旋转,其位置和阴影将发生变化,改变光线的照射方向,其阴影也将随之变化。要做这些工作,可以为添加 CShadowView 类添加成员变量作为交互控制的接口,请读者自己编程完成。

因此,本节所介绍的技术是具有通用性的。

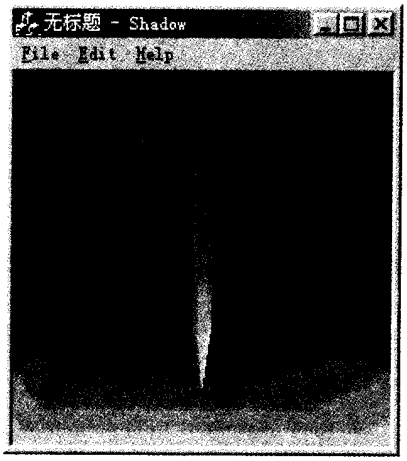

图 16-8 阴影的制作

16.5.3 程序清单

Point.h 文件：

```
//////////////////////////////////////////////////////////////////////////////////////////
//////////////////////////                                  //////////////////////////
//////////////////////////            Point.h               //////////////////////////
//////////////////////////                                  //////////////////////////
//////////////////////////////////////////////////////////////////////////////////////////

// Point.h: interface for the Point class.

#include "gl\gl.h"
#include "math.h"

#if !defined(AFX_POINT_H__B7EFCF01_82AF_11D2_83B4_0080C832FCF3__INCLUDED_)
#define AFX_POINT_H__B7EFCF01_82AF_11D2_83B4_0080C832FCF3__INCLUDED_

#if _MSC_VER >= 1000
#pragma once
#endif // _MSC_VER >= 1000

#define    minLimit (float)0.000001

class Point
{
public:

    Point();
    Point(GLfloat x,GLfloat y,GLfloat z,GLfloat w);
    virtual    ~Point();

public:
    //运算符重载
    inline    Point    operator=(Point a);
    inline    Point    operator=(GLfloat *a);
    inline    Point    operator+(Point a);
    inline    Point    operator+=(Point a);
    inline    Point    operator-(Point a);

    inline    Point    operator*(Point b);
    inline    Point    operator*(GLfloat b);
    inline    Point    operator*=(GLfloat b);
```

```
inline      Point       operator/(GLfloat b);
inline      Point       operator/=(GLfloat b);
inline      GLfloat&    operator[](int index);

inline      void        set(GLfloat x, GLfloat y, GLfloat z, GLfloat w = 1.0);
//向量的各种运算
inline      GLfloat     dist(Point b);
inline      GLfloat     dot(Point b);
inline      GLfloat     dot(GLfloat x, GLfloat y, GLfloat z);
inline      GLfloat     mag();
inline      GLfloat     magsquared();
inline      Point       unit();
inline      void        unitize();

inline      Point       scale(Point p);

// 点的旋转:角度为弧度
inline      Point       rotate(Point axis, GLfloat angle);
inline      Point       rotate(Point axis, GLfloat c, GLfloat s);
inline      void        rotate_self(Point axis, GLfloat c, GLfloat s);

Point       rotate_aboutx(GLfloat c, GLfloat s);
Point       rotate_aboutx(GLfloat angle);
Point       rotate_abouty(GLfloat c, GLfloat s);
Point       rotate_abouty(GLfloat angle);
Point       rotate_aboutz(GLfloat c, GLfloat s);
Point       rotate_aboutz(GLfloat angle);

// 透视到 o_xy 平面
inline      Point       project(Point proj_pt);
inline      void        project_self(Point proj_pt);
inline      void        project_self(GLfloat px, GLfloat py, GLfloat pz);
inline      Point       project_direction(Point direction);
inline      Point       project_direction(GLfloat x, GLfloat y, GLfloat z);

inline      void        compute_projected(GLfloat px, GLfloat py, GLfloat pz,
                        GLfloat x, GLfloat y, GLfloat z);

inline      Point       refract(Point light, Point N, GLfloat I);
void        refract_self(Point light, Point N, GLfloat I);

inline      void        glvertex();
inline      void        glnormal();
```

```
    Point       CalcuNormal(Point point2,Point point3);                    //法线的计算
    Point       refract_direction(Point light, Point N, GLfloat I);        //反射

    GLfloat     pt[4];
};

#define DOT(a, b) (a.pt[0] * b.pt[0] + a.pt[1] * b.pt[1] + a.pt[2] * b.pt[2])
#define THIS_DOT(b) (pt[0] * b.pt[0] + pt[1] * b.pt[1] + pt[2] * b.pt[2])

inline Point Point::operator = (Point a)
{
    pt[0] = a.pt[0];
    pt[1] = a.pt[1];
    pt[2] = a.pt[2];
    pt[3] = a.pt[3];
    return * this;
}

inline Point Point::operator = (GLfloat * a)
{
    pt[0] = a[0];
    pt[1] = a[1];
    pt[2] = a[2];
    pt[3] = 1;
    return * this;
}

//向量加法
inline Point Point::operator + (Point a)
{
    Point resultPt;
    resultPt.pt[0] = pt[0] + a.pt[0];
    resultPt.pt[1] = pt[1] + a.pt[1];
    resultPt.pt[2] = pt[2] + a.pt[2];
    return resultPt;
}

inline Point Point::operator + = (Point a)
{
    pt[0] + = a.pt[0];
    pt[1] + = a.pt[1];
    pt[2] + = a.pt[2];
    return * this;
}
```

```
//向量减法
inline Point Point::operator-(Point a)
{
	Point resultPt;
	resultPt.pt[0] = pt[0] - a.pt[0];
	resultPt.pt[1] = pt[1] - a.pt[1];
	resultPt.pt[2] = pt[2] - a.pt[2];
	return resultPt;
}

//向量积
inline Point Point::operator*(Point b)
{
	Point resultPt;
	resultPt.pt[0] = pt[1]*b.pt[2] - b.pt[1]*pt[2];
	resultPt.pt[1] = pt[2]*b.pt[0] - b.pt[2]*pt[0];
	resultPt.pt[2] = pt[0]*b.pt[1] - pt[1]*b.pt[0];
	return resultPt;
}

//标量与向量积
inline Point Point::operator*(GLfloat b)
{
	Point resultPt;
	resultPt.pt[0] = pt[0] * b;
	resultPt.pt[1] = pt[1] * b;
	resultPt.pt[2] = pt[2] * b;
	return resultPt;
}

inline Point Point::operator*=(GLfloat b)
{
	pt[0] *= b;
	pt[1] *= b;
	pt[2] *= b;
	return *this;
}

inline Point Point::operator/(GLfloat b)
{
	Point resultPt;
	resultPt.pt[0] = pt[0] / b;
	resultPt.pt[1] = pt[1] / b;
```

```
        resultPt.pt[2] = pt[2] / b;
        return resultPt;
}

inline Point Point::operator/=(GLfloat b)
{
        pt[0] /= b;
        pt[1] /= b;
        pt[2] /= b;
        return *this;
}

inline GLfloat& Point::operator[](int index)
{
        return pt[index];
}

//设置一个向量
inline void Point::set(GLfloat x, GLfloat y, GLfloat z, GLfloat w)
{
        pt[0] = x;
        pt[1] = y;
        pt[2] = z;
        pt[3] = w;
}

//两点间的距离
inline GLfloat Point::dist(Point b)
{
        return (*this - b).mag();
}

//向量点积
inline GLfloat Point::dot(Point b)
{
        return pt[0]*b.pt[0] + pt[1]*b.pt[1] + pt[2]*b.pt[2];
}

inline GLfloat Point::dot(GLfloat x, GLfloat y, GLfloat z)
{
        return pt[0]*x + pt[1]*y + pt[2]*z;
}

//向量的范数
```

```
inline GLfloat Point::mag()
{
    return sqrt(pt[0] * pt[0] + pt[1] * pt[1] + pt[2] * pt[2]);
}

inline GLfloat Point::magsquared()
{
    return pt[0] * pt[0] + pt[1] * pt[1] + pt[2] * pt[2];
}

//向量的单位化
inline Point Point::unit()
{
    Point resultPt;
    GLfloat m = sqrt(pt[0] * pt[0] + pt[1] * pt[1] + pt[2] * pt[2]);
    resultPt.pt[0] = pt[0] / m;
    resultPt.pt[1] = pt[1] / m;
    resultPt.pt[2] = pt[2] / m;
    return resultPt;
}

inline void Point::unitize()
{
    GLfloat m = sqrt(pt[0] * pt[0] + pt[1] * pt[1] + pt[2] * pt[2]);
    pt[0] /= m;
    pt[1] /= m;
    pt[2] /= m;
}

inline Point Point::scale(Point p)
{
    Point resultPt;
    resultPt.pt[0] = pt[0] * p.pt[0];
    resultPt.pt[1] = pt[1] * p.pt[1];
    resultPt.pt[2] = pt[2] * p.pt[2];
    return resultPt;
}

//向量绕轴的旋转:点的旋转
inline Point Point::rotate(Point axis, GLfloat angle)
{
    return rotate(axis, cos(angle), sin(angle));
}
```

```
inline Point Point::rotate(Point axis, GLfloat c, GLfloat s)
{
    Point resultPt;

    axis = axis.unit(); //单位化
    float x = axis.pt[0], y = axis.pt[1], z = axis.pt[2], t = 1.0 - c;
    float tx, ty;
    tx = t * x;

    //通用情形
    if (y == 0.0)
    {
        resultPt.pt[0] = pt[0] * (tx * x + c) + pt[1] * (-s * z) + pt[2] * (tx * z);
        resultPt.pt[1] = pt[0] * (s * z) + pt[1] * c + pt[2] * (-s * x);
        resultPt.pt[2] = pt[0] * (tx * z) + pt[1] * s * x + pt[2] * (t * z * z + c);
    }
    else
    {
        ty = t * y;
        resultPt.pt[0] = pt[0] * (tx * x + c) + pt[1] * (tx * y - s * z) + pt[2] * (tx * z + s * y);
        resultPt.pt[1] = pt[0] * (tx * y + s * z) + pt[1] * (ty * y + c) + pt[2] * (ty * z - s * x);
        resultPt.pt[2] = pt[0] * (tx * z - s * y) + pt[1] * (ty * z + s * x) + pt[2] * (t * z * z + c);
    }
    return resultPt;
}

inline void Point::rotate_self(Point axis, GLfloat c, GLfloat s)
{
    float Px, Py, Pz;
    float x = axis.pt[0], y = axis.pt[1], z = axis.pt[2], t = 1.0 - c;
    float tx, ty;

    tx = t * x;
    Px = pt[0];
    Py = pt[1];
    Pz = pt[2];

    if (! y)
    {
        pt[0] = Px * (tx * x + c) + Py * (-s * z) + Pz * (tx * z);
        pt[1] = Px * (s * z) + Py * c + Pz * (-s * x);
        pt[2] = Px * (tx * z) + Py * s * x + Pz * (t * z * z + c);
    }
    else
```

```
    {
        ty = t * y;
        pt[0] = Px * (tx * x + c) + Py * (tx * y - s * z) + Pz * (tx * z + s * y);
        pt[1] = Px * (tx * y + s * z) + Py * (ty * y + c) + Pz * (ty * z - s * x);
        pt[2] = Px * (tx * z - s * y) + Py * (ty * z + s * x) + Pz * (t * z * z + c);
    }
}

inline void Point::glvertex()
{
    glVertex3fv(pt);
}

inline void Point::glnormal()
{
    glNormal3fv(pt);
}

//点的透视
inline Point Point::project(Point proj_pt)
{
    Point resultPt;
    GLfloat dirx = pt[0] - proj_pt.pt[0],
      diry = pt[1] - proj_pt.pt[1],
      dirz = pt[2] - proj_pt.pt[2];
    GLfloat t;

    if (fabs(dirz) < minLimit) resultPt = * this;
    else
    {
        t = - proj_pt.pt[2] / dirz;
        resultPt.pt[0] = proj_pt.pt[0] + dirx * t;
        resultPt.pt[1] = proj_pt.pt[1] + diry * t;
        resultPt.pt[2] = 0.0;
    }
    return resultPt;
}

inline void Point::project_self(Point proj_pt)
{
    GLfloat dirx = pt[0] - proj_pt.pt[0],
      diry = pt[1] - proj_pt.pt[1],
      dirz = pt[2] - proj_pt.pt[2];
    GLfloat t;
```

```
    t = -proj_pt.pt[2] / dirz;
    pt[0] = proj_pt.pt[0] + dirx * t;
    pt[1] = proj_pt.pt[1] + diry * t;
    pt[2] = 0.0;
}

inline void Point::project_self(GLfloat px, GLfloat py, GLfloat pz)
{
    GLfloat dirx = pt[0] - px,
    diry = pt[1] - py,
    dirz = pt[2] - pz, t;

    t = -pz / dirz;
    pt[0] = px + dirx * t;
    pt[1] = py + diry * t;
    pt[2] = 0.0;
}

//求在给定透视方向上的透视点
inline Point Point::project_direction(Point direction)
{
    Point resultPt;
    GLfloat t;

    t = -pt[2] / direction.pt[2];
    resultPt.pt[0] = pt[0] + direction.pt[0] * t;
    resultPt.pt[1] = pt[1] + direction.pt[1] * t;
    resultPt.pt[2] = 0;
    return resultPt;
}

inline Point Point::project_direction(GLfloat x, GLfloat y, GLfloat z)
{
    Point resultPt;
    GLfloat t;

    t = -pt[2] / z;
    resultPt.pt[0] = pt[0] + x * t;
    resultPt.pt[1] = pt[1] + y * t;
    resultPt.pt[2] = 0;
    return resultPt;
}
```

```
inline void Point::compute_projected(GLfloat px,GLfloat py, GLfloat pz,
                                     GLfloat dx, GLfloat dy, GLfloat dz)
{
    GLfloat t = -pz / dz;
    pt[0] = px + dx * t;
    pt[1] = py + dy * t;
    pt[2] = 0;
}

#endif // ! defined(AFX_POINT_H__B7EFCF01_82AF_11D2_83B4_0080C832FCF3__INCLUDED_)
```

Point.cpp 文件:

```
////////////////////////////////////////////////////////////////////////////////////////////
//////////////////////////                                  //////////////////////////
//////////////////////////          Point.cpp               //////////////////////////
//////////////////////////                                  //////////////////////////
////////////////////////////////////////////////////////////////////////////////////////


// Point.cpp: implementation of the Point class.
//
//////////////////////////////////////////////////////////////////////

#include "stdafx.h"
#include "Shadow.h"
#include "Point.h"

#ifdef _DEBUG
#undef THIS_FILE
static char THIS_FILE[]=__FILE__;
#define new DEBUG_NEW
#endif

//////////////////////////////////////////////////////////////////////
// Construction/Destruction
//////////////////////////////////////////////////////////////////////

Point::Point()
{
}

Point::Point(GLfloat x,GLfloat y,GLfloat z,GLfloat w)
{
```

```
    pt[0] = x;
    pt[1] = y;
    pt[2] = z;
    pt[3] = w;
}

Point::~Point()
{
}

//绕 x 轴
Point Point::rotate_aboutx(GLfloat c, GLfloat s)
{
    Point resultPt;

    resultPt.pt[0] = pt[0];
    resultPt.pt[1] = c * pt[1] - s * pt[2];
    resultPt.pt[2] = s * pt[1] + c * pt[2];
    return resultPt;
}

Point Point::rotate_aboutx(GLfloat angle)
{
    Point resultPt;
    GLfloat c = cos(angle);
    GLfloat s = sin(angle);

    resultPt.pt[0] = pt[0];
    resultPt.pt[1] = c * pt[1] - s * pt[2];
    resultPt.pt[2] = s * pt[1] + c * pt[2];
    return resultPt;
}

//绕 y 轴
Point Point::rotate_abouty(GLfloat c, GLfloat s)
{
    Point resultPt;
    resultPt.pt[0] = c * pt[0] + s * pt[2];
    resultPt.pt[1] = pt[1];
    resultPt.pt[2] = -s * pt[0] + c * pt[2];
    return resultPt;
}

Point Point::rotate_abouty(GLfloat angle)
```

```
{
    Point resultPt;
    GLfloat c = cos(angle);
    GLfloat s = sin(angle);

    resultPt.pt[0] = c * pt[0] + s * pt[2];
    resultPt.pt[1] = pt[1];
    resultPt.pt[2] = -s * pt[0] + c * pt[2];
    return resultPt;
}

//绕 z 轴
Point Point::rotate_aboutz(GLfloat c, GLfloat s)
{
    Point resultPt;
    resultPt.pt[0] = c * pt[0] - s * pt[1];
    resultPt.pt[1] = s * pt[0] + c * pt[1];
    resultPt.pt[2] = pt[2];
    return resultPt;
}

Point Point::rotate_aboutz(GLfloat angle)
{
    Point resultPt;
    GLfloat c = cos(angle);
    GLfloat s = sin(angle);

    resultPt.pt[0] = c * pt[0] - s * pt[1];
    resultPt.pt[1] = s * pt[0] + c * pt[1];
    resultPt.pt[2] = pt[2];
    return resultPt;
}

//光反射
void Point::refract_self(Point light, Point N, GLfloat I)
{
    GLfloat t;
    Point dlight;

    dlight = refract_direction(light, N, I);
    t = -pt[2] / dlight.pt[2];
    pt[0] = pt[0] + dlight.pt[0] * t;
    pt[1] = pt[1] + dlight.pt[1] * t;
    pt[2] = 0;
```

Point Point::refract_direction(Point light, Point N, GLfloat I)

```
Point resultPt;
GLfloat cos1, sin1, cos2, sin2, m;
GLfloat dlight[3], dN[3], axis[3];

dlight[0] = pt[0] - light.pt[0];
dlight[1] = pt[1] - light.pt[1];
dlight[2] = pt[2] - light.pt[2];
m = sqrt(dlight[0] * dlight[0] + dlight[1] * dlight[1] + dlight[2] * dlight[2]);
dlight[0] /= m;
dlight[1] /= m;
dlight[2] /= m;

// dN = N * -1.0;
dN[0] = -N.pt[0];
dN[1] = -N.pt[1];
dN[2] = -N.pt[2];

cos1 = dN[0] * dlight[0] + dN[1] * dlight[1] + dN[2] * dlight[2];
if (1.0 - cos1 * cos1 < minLimit)
{
    resultPt = dN;
    return resultPt;
}

resultPt.pt[0] = dN[1] * dlight[2] - dlight[1] * dN[2];
resultPt.pt[1] = dN[2] * dlight[0] - dlight[2] * dN[0];
resultPt.pt[2] = dN[0] * dlight[1] - dN[1] * dlight[0];
axis[0] = resultPt.pt[1] * dN[2] - dN[1] * resultPt.pt[2];
axis[1] = resultPt.pt[2] * dN[0] - dN[2] * resultPt.pt[0];
axis[2] = resultPt.pt[0] * dN[1] - resultPt.pt[1] * dN[0];
m = sqrt(axis[0] * axis[0] + axis[1] * axis[1] + axis[2] * axis[2]);
axis[0] /= m;
axis[1] /= m;
axis[2] /= m;

if (axis[0] * axis[0] > minLimit)
sin1 = (dlight[0] - dN[0] * cos1) / axis[0];
else if (axis[1] * axis[1] > minLimit)
sin1 = (dlight[1] - dN[1] * cos1) / axis[1];
else sin1 = dlight[2] - dN[2] * cos1;
```

```
        sin2 = sin1 / I;
        cos2 = (sin1 * sin1 < 1.0) ? sqrt(1.0 - sin2 * sin2) : 0;

        dlight[0] = dN[0] * cos2 + axis[0] * sin2;
        dlight[1] = dN[1] * cos2 + axis[1] * sin2;
        dlight[2] = dN[2] * cos2 + axis[2] * sin2;
        if (dlight[2] > 0.0) dlight[2] = -dlight[2];

        resultPt = dlight;
        return resultPt;
}

//法线的计算
Point Point::CalcuNormal(Point point2, Point point3)
{
        Point resultPt;
        Point vector1,vector2,point1;

        point1.pt[0] = pt[0];
        point1.pt[1] = pt[1];
        point1.pt[2] = pt[2];

        vector1 = point2 - point1;
        vector2 = point3 - point1;

        resultPt = vector1 * vector2;

        resultPt.unitize();
        return resultPt;
```

Color.h 文件：

```
////////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////                                  ////////////////////////////
////////////////////////////          Color.h                 ////////////////////////////
////////////////////////////                                  ////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////


// Color.h: interface for the Color class.

# include "gl \ gl.h"
```

```
#if !defined(AFX_COLOR_H__27EED6A2_8BAE_11D2_83B4_0080C832FCF3__INCLUDED_)
#define AFX_COLOR_H__27EED6A2_8BAE_11D2_83B4_0080C832FCF3__INCLUDED_

#if _MSC_VER >= 1000
#pragma once
#endif // _MSC_VER >= 1000

class Color
{
public:
	Color();
	virtual ~Color();
public:
	inline	Color(GLfloat r, GLfloat g, GLfloat b, GLfloat a = 1.0);
	inline	Color operator=(GLfloat *a);
	inline	Color operator=(GLfloat a);
	inline	Color operator=(Color color);
	inline	Color operator+(Color a);
	inline	Color operator+=(Color a);
	inline	Color operator*(Color a);
	inline	Color operator*(GLfloat a);
	inline	Color operator*=(Color a);
	inline	Color operator*=(GLfloat *a);
	inline	Color operator*=(GLfloat a);

	inline	GLfloat& operator[](int index);

	inline	Color clamp();

	inline	void glcolor();

	void set(GLfloat r,GLfloat g,GLfloat b,GLfloat a=1.0);

public:

	GLfloat c[4];
};

inline Color::Color(GLfloat r, GLfloat g, GLfloat b, GLfloat a)
{
	c[0] = r;
	c[1] = g;
	c[2] = b;
	c[3] = a;
```

```
}

inline Color Color::operator=(GLfloat a)
{
        c[0] = c[1] = c[2] = c[3] = a;
        return *this;
}

inline Color Color::operator=(GLfloat *a)
{
        c[0] = a[0];
        c[1] = a[1];
        c[2] = a[2];
        c[3] = a[3];
        return *this;
}

inline Color Color:: operator=(Color color)
{
        c[0] = color.c[0];
        c[1] = color.c[1];
        c[2] = color.c[2];
        c[3] = color.c[3];
        return *this;
}

inline Color Color::operator+(Color a)
{
        Color resultCl;
        resultCl.c[0] = c[0] + a.c[0];
        resultCl.c[1] = c[1] + a.c[1];
        resultCl.c[2] = c[2] + a.c[2];
        resultCl.c[3] = c[3] + a.c[3];
        return resultCl;
}

inline Color Color::operator+=(Color a)
{
        c[0] += a.c[0];
        c[1] += a.c[1];
        c[2] += a.c[2];
        c[3] += a.c[3];
        return *this;
}
```

```
inline Color Color::operator * (Color a)
{
    Color resultCl;
    resultCl.c[0] = c[0] * a.c[0];
    resultCl.c[1] = c[1] * a.c[1];
    resultCl.c[2] = c[2] * a.c[2];
    resultCl.c[3] = c[3] * a.c[3];
    return resultCl;
}

inline Color Color::operator * (GLfloat a)
{
    Color resultCl;
    resultCl.c[0] = c[0] * a;
    resultCl.c[1] = c[1] * a;
    resultCl.c[2] = c[2] * a;
    resultCl.c[3] = c[3] * a;
    return resultCl;
}

inline Color Color::operator * = (Color a)
{
    c[0] * = a.c[0];
    c[1] * = a.c[1];
    c[2] * = a.c[2];
    return * this;
}

inline Color Color::operator * = (GLfloat * a)
{
    c[0] * = a[0];
    c[1] * = a[1];
    c[2] * = a[2];
    return * this;
}

inline Color Color::operator * = (GLfloat a)
{
    c[0] * = a;
    c[1] * = a;
    c[2] * = a;
    c[3] * = a;
```

```
    return * this;
}

inline GLfloat& Color::operator[](int index)
{
    return c[index];
}

//颜色分量的归一化
inline Color Color::clamp()
{
    Color resultCl;
    resultCl.c[0] = c[0] < 0.0 ? 0.0 : (c[0] > 1.0 ? 1.0 : c[0]);
    resultCl.c[1] = c[1] < 0.0 ? 0.0 : (c[1] > 1.0 ? 1.0 : c[1]);
    resultCl.c[2] = c[2] < 0.0 ? 0.0 : (c[2] > 1.0 ? 1.0 : c[2]);
    resultCl.c[3] = c[3] < 0.0 ? 0.0 : (c[3] > 1.0 ? 1.0 : c[3]);
    return resultCl;
}

inline void Color::glcolor()
{
    glColor4fv(c);
}

#endif // ! defined(AFX_COLOR_H__27EED6A2_8BAE_11D2_83B4_0080C832FCF3__
                INCLUDED_)
```

Color.cpp 文件：

```
////////////////////////////////////////////////////////////////////////////////////////////////
//////////////////////////                                  //////////////////////////
//////////////////////////          Color.cpp               //////////////////////////
//////////////////////////                                  //////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////////
// Color.cpp: implementation of the Color class.

#include "stdafx.h"
#include "Shadow.h"
#include "Color.h"

#ifdef _DEBUG
#undef THIS_FILE
static char THIS_FILE[]=__FILE__;
#define new DEBUG_NEW
#endif
```

```
//////////////////////////////////////////////////////////////////////
// Construction/Destruction
//////////////////////////////////////////////////////////////////////

Color::Color()
{
	c[0] = 1.0;
	c[1] = 1.0;
	c[2] = 1.0;
	c[3] = 1.0;
}

Color::~Color()
{
}

void Color::set(GLfloat r, GLfloat g, GLfloat b, GLfloat a)
{
	c[0] = r;
	c[1] = g;
	c[2] = b;
	c[3] = a;
}
```

Pencil.h 文件：

```
//////////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////                                    ////////////////////////////
////////////////////////////          Pencil.h                  ////////////////////////////
////////////////////////////                                    ////////////////////////////
//////////////////////////////////////////////////////////////////////////////////////////////

// Pencil.h: interface for the Pencil class.

#include "Point.h"
#include "Color.h"

#if !defined(AFX_PENCIL_H__CF4EB3C1_8956_11D2_83B4_0080C832FCF3__INCLUDED_)
#define AFX_PENCIL_H__CF4EB3C1_8956_11D2_83B4_0080C832FCF3__INCLUDED_

#if _MSC_VER >= 1000
#pragma once
#endif // _MSC_VER >= 1000
```

```
class Pencil
{
public:
        Pencil();
        virtual             ~Pencil();
        Pencil              operator=(Pencil pencil);
public:

        void        set_shadow_penu_mat(GLfloat* shad_clr, GLfloat* penu_clr);
        void        set_neb_cap_mat(GLfloat* a, GLfloat* d, GLfloat* s);
        void        set_holder_mat(GLfloat* a, GLfloat* d, GLfloat* s);
        void        draw_shadow(Point sun_direction, GLfloat blur);
        void        draw_shadow(Point sun_direction);
        void        draw();

        void        rotate_angle_z(GLfloat anglez);
        void        rotate_angle_y(GLfloat angley);
        void        rotate_angle_x(GLfloat anglex);
        void        translate_parameter(Point transPt);
        void        translate_parameter(GLfloat x,GLfloat y,GLfloat z);
        float       get_erosion();
        void        erode(float eroding);

        Point       get_dimensions();
        void        set_dimensions(Point p);
        void        set_dimensions(GLfloat x, GLfloat y, GLfloat z);
        void        ResetPencil();
        void        SetPencil(GLfloat fRadius, GLfloat zHeight1, GLfloat zHeight2);

private:

        void        material_penumbra();
        void        material_shadow();
        void        material_cap_neb();
        void        material_holder();

        void        after_transform_Ns(Point* resultNs);
        void        GeneratePoints();
        void        GenerateNormals();
        void        draw_holder_cap_neb(Point* p,int flat=0);

        void        transform_points(Point* p);
        void        transform_pencilPts();
        inline      Point transform_point(Point point);
```

```
    void        draw_holder_cap_neb(int flat = 0);
    void        draw_triangle(Point p1,Point p2,Point p3);
    void        draw_quads(Point p1, Point p2, Point p3,Point p4,int shadow_penumbra);
    void        draw_quads(Point p1,Point p2,Point p3,Point p4);

private:

    Point PencilPs[13];                    //铅笔的顶点
    Point PencilNs[13];                    //铅笔面的法线信息

    //铅笔的几何位置及几何尺寸的变换
    Point translation;
    Point dimensions;
    GLfloat     rotation_x, rotation_y, rotation_z;

    GLfloat     radius;                    //笔粗
    //笔长度
    GLfloat     zHeightB;                  //Bottom
    GLfloat     zHeightT;                  //Top

    //是否采用高级半阴技术,点的有效性、位置是否经过几何变换。
    GLfloat     erosion;
    int         points_valid;
    int         transforms_valid;

    //笔的颜色:a:环境,d:漫反射,s:高亮色
    Color holder_a,       holder_d,       holder_s;
    Color neb_cap_a,      neb_cap_d,      neb_cap_s;
    //阴影和半阴色
    Color                 shadow_c,       penu_c;
};

#endif // ! defined(AFX_PENCIL_H__CF4EB3C1_8956_11D2_83B4_0080C832FCF3__INCLUDED_)
```

Pencil.cpp 文件：

```
////////////////////////////////////////////////////////////////////////////////////////////////
//////////////////////////                                    //////////////////////////
//////////////////////////            Pencil.cpp              //////////////////////////
//////////////////////////                                    //////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////////

// Pencil.cpp: implementation of the Pencil class.
```

```
# include "stdafx.h"
# include "Shadow.h"
# include "Pencil.h"

# ifdef _ DEBUG
# undef THIS _ FILE
static char THIS _ FILE[] = __ FILE __;
# define new DEBUG _ NEW
# endif

//////////////////////////////////////////////////////////////////////
// Construction/Destruction
//////////////////////////////////////////////////////////////////////

# define PI     3.14159265358979323846

inline     float     radians(float a)     {return PI * a / 180.0; }

inline     float     sign(float a)     {return (a > 0. ? 1 : (a < 0. ? -1 : 0)); }

Pencil::Pencil()
{
        radius = 0.1;
        zHeightB = 0.1;
        zHeightT = 0.5;

        translation.set(0, 0, 0);
        rotation_x = 0;
        rotation_y = 0;
        rotation_z = 0;
        dimensions.set(1, 1, 1);

        erosion = 0;

        points_valid = 0;
        transforms_valid = 0;

        holder_a.set(0.3,0.3,0.3);
        holder_d.set(1.0,0.15,0.0);
        holder_s.set(0.9,0.9,0.9);
        neb_cap_a.set(0.3,0.3,0.3);
        neb_cap_d.set(0.55,0.3,0.25);
        neb_cap_s.set(0.7,0.7,0.7);
        shadow_c.set(0.05,0.05,0.05,0.95);
```

```
        penu_c.set(0.15,0.15,0.15,0.75);
}

Pencil::~Pencil()
{
}

//具体的笔和笔间的数据传递
Pencil Pencil::operator=(Pencil pencil)
{
        int i;
        //传递几何位置与尺寸大小
        translation = pencil.translation;
        rotation_x = pencil.rotation_x;
        rotation_y = pencil.rotation_y;
        rotation_z = pencil.rotation_z;
        dimensions = pencil.dimensions;
        radius=pencil.radius;
        zHeightB=pencil.zHeightB;
        zHeightT=pencil.zHeightT;
        //模糊因子:即 blur 因子
        erosion = pencil.erosion;
        //点的有效性检验与传递
        if (pencil.points_valid && pencil.transforms_valid)
        {
            for (i = 0; i < 13; i++)
            {
                PencilPs[i]=pencil.PencilPs[i];
                PencilNs[i]=pencil.PencilNs[i];
            }
        }
        else
        {
            points_valid = 0;
            transforms_valid = 0;
        }
        return *this;
}

//本函数创建铅笔的几何顶点
void Pencil::GeneratePoints()
{
        float c, s, x, y;
        float fAngle=-30.0;
```

```
    int i=0;
    //笔尖顶点为(0,0,0)
    PencilPs[0].set(0,0,0);
    //创建第零至第十二个点
    for(i=1;i<7;i++)
    {
        c = cos(radians(fAngle));
        s = sin(radians(fAngle));
        x=radius*c;
        y=radius*s;
        PencilPs[i].set(x,y,zHeightB);
        PencilPs[i+6].set(x,y,zHeightT);
        fAngle+=60.0;
    }
}

/*创建铅笔表面的法线:未经变换的铅笔,即位于缺省的位置,缺省位置为笔尖在坐标原点,垂直于地面并与z轴平行*/
void Pencil::GenerateNormals()
{
    float x, y;
    float fAngle=0.0;
    int i=0;
    //第六至第十一号面的法线,依次为代表x轴正向的单位向量绕z轴逆时针旋转60度
    for(i=0;i<6;i++)
    {
        x = cos(radians(fAngle));
        y = sin(radians(fAngle));
        PencilNs[i+6].set(x,y,0.0);
        fAngle+=60.0;
    }
    //第零至第五号面的法线则需要通过计算:这些面为三角形
    PencilNs[0]=PencilPs[1].CalcuNormal(PencilPs[0],PencilPs[2]);
    PencilNs[1]=PencilPs[2].CalcuNormal(PencilPs[0],PencilPs[3]);
    PencilNs[2]=PencilPs[3].CalcuNormal(PencilPs[0],PencilPs[4]);
    PencilNs[3]=PencilPs[4].CalcuNormal(PencilPs[0],PencilPs[5]);
    PencilNs[4]=PencilPs[5].CalcuNormal(PencilPs[0],PencilPs[6]);
    PencilNs[5]=PencilPs[6].CalcuNormal(PencilPs[0],PencilPs[1]);
    //铅笔顶面的法线与z轴平行且与其正向同向
    PencilNs[12].set(0.0,0.0,1.0);
}

//变换后的法线计算
```

```
void Pencil::after_transform_Ns(Point * resultNs)
{
    resultNs[0] = PencilPs[1].CalcuNormal(PencilPs[0],PencilPs[2]);
    resultNs[1] = PencilPs[2].CalcuNormal(PencilPs[0],PencilPs[3]);
    resultNs[2] = PencilPs[3].CalcuNormal(PencilPs[0],PencilPs[4]);
    resultNs[3] = PencilPs[4].CalcuNormal(PencilPs[0],PencilPs[5]);
    resultNs[4] = PencilPs[5].CalcuNormal(PencilPs[0],PencilPs[6]);
    resultNs[5] = PencilPs[6].CalcuNormal(PencilPs[0],PencilPs[1]);

    resultNs[6] = PencilPs[1].CalcuNormal(PencilPs[2],PencilPs[8]);
    resultNs[7] = PencilPs[2].CalcuNormal(PencilPs[3],PencilPs[9]);
    resultNs[8] = PencilPs[3].CalcuNormal(PencilPs[4],PencilPs[10]);
    resultNs[9] = PencilPs[4].CalcuNormal(PencilPs[5],PencilPs[11]);
    resultNs[10] = PencilPs[5].CalcuNormal(PencilPs[6],PencilPs[12]);
    resultNs[11] = PencilPs[6].CalcuNormal(PencilPs[1],PencilPs[7]);

    resultNs[12] = PencilPs[7].CalcuNormal(PencilPs[9],PencilPs[11]);
}

//该函数设置铅笔的几何参数
/*
//铅笔由笔杆、笔尖和笔帽构成,笔杆由六个面构成,即一个六棱柱,笔尖由一个六个面组成的六棱锥构成,
//而笔帽则由一个六边形构成。
//笔尖顶点为第零号顶点,第一至第六个顶点则位于一个圆上,该圆的半径为 radius,距离 xoy 平面高度为
//zHeightB;第七至第十二个顶点也位于一个圆上,距离 xoy 平面高度为 zHeightT。第一个顶点是从与 x 轴
//的夹角为 -30 度的圆周上的点的位置开始的,第七个顶
//点也是如此开始的,然后按逆时针方向产生后继顶点。
//基于这样的顶点,第零面的顶点序号为 1,2,0;第一个面的序号为 2,3,0;依次类推;第六个面至第十一个
//面为四边形,第六个面的顶点为 1,7,8,2;第七个面的顶点为 2,8,9,3;依次类推;第十二号面则为顶面,是
//一个六边形。
*/
void Pencil::SetPencil(GLfloat fRadius, GLfloat zHeight1, GLfloat zHeight2)
{
    GLfloat fTemp;
    if(zHeight1 > zHeight2)
    {
        fTemp = zHeight1;
        zHeight1 = zHeight2;
        zHeight2 = fTemp;
    }
    radius = fRadius;
    zHeightB = zHeight1;
    zHeightT = zHeight2;
```

```
        GeneratePoints();
        GenerateNormals();
}

//将铅笔的预定信息赋给经过变换后的铅笔:即均采用缺省值
void Pencil::ResetPencil()
{
        translation.set(0, 0, 0);
        rotation_x = 0;
        rotation_y = 0;
        rotation_z = 0;
        //尺寸保持不变,与初始值同
        dimensions.set(1, 1, 1);
        //不模糊,与初始值同
        erosion = 0;
        //后继变换有效,与初始值同
        points_valid = 0;
        transforms_valid = 0;
        //重新生成顶点和法线矢量
        GeneratePoints();
        GenerateNormals();
}

//设置铅笔的缩放尺寸
void Pencil::set_dimensions(GLfloat x, GLfloat y, GLfloat z)
{
        dimensions.set(x, y, z);
        points_valid = 0;
}

void Pencil::set_dimensions(Point p)
{
        dimensions = p;
        points_valid = 0;
}

Point Pencil::get_dimensions()
{
        return dimensions;
}

//设置半阴影的边缘模糊程度
void Pencil::erode(float eroding)
{
```

```
        erosion += eroding;
        points_valid = 0;
}

float Pencil::get_erosion()
{
        return erosion;
}

//点的平移变换
void Pencil::translate_parameter(GLfloat x, GLfloat y, GLfloat z)
{
        translation.pt[0] += x;
        translation.pt[1] += y;
        translation.pt[2] += z;
        points_valid=0;
}

void Pencil::translate_parameter(Point transPt)
{
        translation += transPt;
        points_valid=0;
}

//绕 x 轴旋转弧度
void Pencil::rotate_angle_x(GLfloat anglex)
{
        rotation_x += anglex;
        points_valid=0;
}

//绕 y 轴旋转弧度
void Pencil::rotate_angle_y(GLfloat angley)
{
        rotation_y += angley;
        points_valid=0;
}

//绕 z 轴旋转弧度
void Pencil::rotate_angle_z(GLfloat anglez)
{
        rotation_z += anglez;
        points_valid=0;
}
```

```
//变换单个点:缩放? 平移? 旋转
Point Pencil::transform_point(Point point)
{
        Point resultPt,tempPt;
        tempPt = point.scale(dimensions);
        tempPt += translation;
        resultPt = tempPt.rotate_aboutz(rotation_z);
        tempPt = resultPt.rotate_abouty(rotation_y);
        resultPt = tempPt.rotate_aboutx(rotation_x);
        return resultPt;
}

//变换铅笔
void Pencil::transform_pencilPts()
{
        int i;
        //如果已经变换就不作任何工作
        if (points_valid && transforms_valid) return;
        points_valid = 0;
        //再次生成点
        GeneratePoints();
        //逐个变换顶点
        for (i = 0; i < 13; i++)
          PencilPs[i] = transform_point(PencilPs[i]);

        //after_transform_Ns();//此时法线可以不必变换

        points_valid = 0;
        transforms_valid = 1;
}

//通用的变换函数
void Pencil::transform_points(Point * p)
{
        for (int i = 0; i < 13; i++)
          p[i] = transform_point(p[i]);
}

//铅笔的绘制
void Pencil::draw()
{
        //保证点的正确性
        GeneratePoints();
```

```
        //一旦调用该函数,顶点值和法线向量值将是变换后的值。
        //先缩放,后平移,再绕 z,y,x 轴旋转。
        transform _ pencilPts();
        after _ transform _ Ns(PencilNs);
        draw _ holder _ cap _ neb();
}

//绘制四边形
void Pencil::draw _ quads(Point p1, Point p2, Point p3, Point p4)
{
        glBegin(GL _ QUADS);
            glVertex3fv(p1.pt);
            glVertex3fv(p2.pt);
            glVertex3fv(p3.pt);
            glVertex3fv(p4.pt);
        glEnd();
}

//绘制四边形
void Pencil::draw _ quads(Point p1, Point p2, Point p3, Point p4,
                    int shadow _ penumbra)
{
        //设置 shadow _ penumbra 为 1,以及其他任何一个数都可以
        shadow _ penumbra = 1;
        glBegin(GL _ QUADS);
            material _ shadow();
            glVertex3fv(p1.pt);
            glVertex3fv(p2.pt);

            material _ penumbra();
            glVertex3fv(p3.pt);
            glVertex3fv(p4.pt);
        glEnd();
}

//绘制三角形
void Pencil::draw _ triangle(Point p1,Point p2,Point p3)
{
        glBegin(GL _ TRIANGLES);
            glVertex3fv(p1.pt);
            glVertex3fv(p2.pt);
            glVertex3fv(p3.pt);
        glEnd();
```

```
}

//绘制阴影
void Pencil::draw_shadow(Point sun_direction)
{
    Point p[13];
    int i;

    //    transform_pencilPts();
    //由于先绘制铅笔,所以,不必进行变换。
    for(i=0;i<13;i++)
    {
        p[i]=PencilPs[i];
        if(p[i].pt[2]<0.0)p[i].pt[2]=0.0;
        p[i]=p[i].project_direction(sun_direction);
    }
    draw_holder_cap_neb(p,1);
}

//绘制笔杆、笔帽和笔尖
void Pencil::draw_holder_cap_neb(Point * p, int flat)
{
    GLfloat fN[]={0.0,0.0,1.0};

    //draw holder
    material_holder();
    if(flat)material_shadow();

    glNormal3fv(PencilNs[6].pt);
    if(flat)glNormal3fv(fN);
    draw_quads(p[1],p[2],p[8],p[7]);

    glNormal3fv(PencilNs[7].pt);
    if(flat)glNormal3fv(fN);
    draw_quads(p[2],p[3],p[9],p[8]);

    glNormal3fv(PencilNs[8].pt);
    if(flat)glNormal3fv(fN);
    draw_quads(p[3],p[4],p[10],p[9]);

    glNormal3fv(PencilNs[9].pt);
    if(flat)glNormal3fv(fN);
    draw_quads(p[4],p[5],p[11],p[10]);
```

```
glNormal3fv(PencilNs[10].pt);
if(flat)glNormal3fv(fN);
draw_quads(p[5],p[6],p[12],p[11]);

glNormal3fv(PencilNs[11].pt);
if(flat)glNormal3fv(fN);
draw_quads(p[6],p[1],p[7],p[12]);

//draw cap
material_cap_neb();
if(flat)material_shadow();

glBegin(GL_POLYGON);
    glNormal3fv(PencilNs[12].pt);
    if(flat)glNormal3f(0.0f,0.0f,1.0f);
    glVertex3fv(p[7].pt);
    glVertex3fv(p[8].pt);
    glVertex3fv(p[9].pt);
    glVertex3fv(p[10].pt);
    glVertex3fv(p[11].pt);
    glVertex3fv(p[12].pt);
glEnd();

//draw neb
glNormal3fv(PencilNs[0].pt);
if(flat)glNormal3fv(fN);
draw_triangle(p[2],p[1],p[0]);

glNormal3fv(PencilNs[1].pt);
if(flat)glNormal3fv(fN);
draw_triangle(p[3],p[2],p[0]);

glNormal3fv(PencilNs[2].pt);
if(flat)glNormal3fv(fN);
draw_triangle(p[4],p[3],p[0]);

glNormal3fv(PencilNs[3].pt);
if(flat)glNormal3fv(fN);
draw_triangle(p[5],p[4],p[0]);

glNormal3fv(PencilNs[4].pt);
if(flat)glNormal3fv(fN);
draw_triangle(p[6],p[5],p[0]);
```

```
        glNormal3fv(PencilNs[5].pt);
        if(flat)glNormal3fv(fN);
        draw_triangle(p[1],p[6],p[0]);
}

void Pencil::draw_holder_cap_neb(int flat)
{
        draw_holder_cap_neb(PencilPs,flat);
}

//绘制阴影:光照射方向,模糊程度、半影的绘制
void Pencil::draw_shadow(Point sun_direction, GLfloat blur)
{
        Point p[13];
        Point ptBlur[13];
        Point n[13];
        Point scale_blur;
        int yes_or_no[13];
        int i;
        //如果 blur 为 0.0,根本就没有半影
        if(blur==0)
        {
            draw_shadow(sun_direction);
            return;
        }
        //生成原始顶点(管理原始顶点可减少重复计算)
        GeneratePoints();
        for(i=0;i<13;i++) ptBlur[i] = PencilPs[i];

        //注意,重载函数 draw_shadow()函数不能同时使用,否则会使顶点进行两次变换
        //假定对铅笔的几何点还没有执行几何变换
        transform_pencilPts();

        //将顶点透视到平面
        for(i=0;i<13;i++)
        {
            p[i] = PencilPs[i];
            if(p[i].pt[2] < 0.0)p[i].pt[2]=0.0;
            p[i] = p[i].project_direction(sun_direction);
            //透视经过变换后的点的半影点
            scale_blur = dimensions;
            scale_blur.pt[0] += blur * (ptBlur[i].pt[2] + translation.pt[2]);
            scale_blur.pt[1] += blur * (ptBlur[i].pt[2] + translation.pt[2]);
            scale_blur.pt[2] += blur * (ptBlur[i].pt[2] + translation.pt[2]);
```

```
        ptBlur[i] = ptBlur[i].scale(scale_blur);
        ptBlur[i] = transform_point(ptBlur[i]);
        if(ptBlur[i].pt[2] < 0.0)ptBlur[i].pt[2] = 0.0;
        ptBlur[i] = ptBlur[i].project_direction(sun_direction);
    }
    //几何变换后的法线:
    after_transform_Ns(n);
    //决定画与不画阴影(背光的一面不画)
    if(sun_direction.pt[2] > 0.0)sun_direction *= -1.0;
    for(i = 0;i < 13;i++)
        yes_or_no[i] = (sun_direction.dot(n[i]) < 0.001);
    //绘制真实阴影
    GLfloat fN[] = {0.0,0.0,1.0};
    glPolygonMode(GL_FRONT_AND_BACK,GL_FILL);
    glColor3f(0.25f,0.25f,0.25f);
    glNormal3fv(fN);

    //neb
    if(yes_or_no[0])
        draw_triangle(p[2],p[1],p[0]);
    if(yes_or_no[1])
        draw_triangle(p[3],p[2],p[0]);
    if(yes_or_no[2])
        draw_triangle(p[4],p[3],p[0]);
    if(yes_or_no[3])
        draw_triangle(p[5],p[4],p[0]);
    if(yes_or_no[4])
        draw_triangle(p[6],p[5],p[0]);
    if(yes_or_no[5])
        draw_triangle(p[1],p[6],p[0]);

    //holder
    if(yes_or_no[6])
        draw_quads(p[1],p[2],p[8],p[7]);
    if(yes_or_no[7])
        draw_quads(p[2],p[3],p[9],p[8]);
    if(yes_or_no[8])
        draw_quads(p[3],p[4],p[10],p[9]);
    if(yes_or_no[9])
        draw_quads(p[4],p[5],p[11],p[10]);
    if(yes_or_no[10])
        draw_quads(p[5],p[6],p[12],p[11]);
    if(yes_or_no[11])
        draw_quads(p[6],p[1],p[7],p[12]);
```

```
//cap
if(yes_or_no[12])
{
    glBegin(GL_POLYGON);
        glVertex3fv(p[7].pt);
        glVertex3fv(p[8].pt);
        glVertex3fv(p[9].pt);
        glVertex3fv(p[10].pt);
        glVertex3fv(p[11].pt);
        glVertex3fv(p[12].pt);
    glEnd();
}
//绘制半影(penumbra),简化的半影绘制技术:边缘线的扩大与阴影的模糊
//绘制半影的技术较复杂,此处作一些简化
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);

//cap
if(yes_or_no[12])
{
    if(! yes_or_no[6])
        draw_quads(p[7],p[8],ptBlur[8],ptBlur[7],1);
    if(! yes_or_no[7])
        draw_quads(p[8],p[9],ptBlur[9],ptBlur[8],1);
    if(! yes_or_no[8])
        draw_quads(p[9],p[10],ptBlur[10],ptBlur[9],1);
    if(! yes_or_no[9])
        draw_quads(p[10],p[11],ptBlur[11],ptBlur[10],1);
    if(! yes_or_no[10])
        draw_quads(p[11],p[12],ptBlur[12],ptBlur[11],1);
    if(! yes_or_no[11])
        draw_quads(p[12],p[7],ptBlur[7],ptBlur[12],1);
}
//holder
if ((yes_or_no[6] || yes_or_no[7])
        && ! (yes_or_no[6] && yes_or_no[7]))
            draw_quads(p[2],p[8],ptBlur[8],ptBlur[2],1);
if ((yes_or_no[7] || yes_or_no[8])
        && ! (yes_or_no[7] && yes_or_no[8]))
            draw_quads(p[3],p[9],ptBlur[9],ptBlur[3],1);
if ((yes_or_no[8] || yes_or_no[9])
        && ! (yes_or_no[8] && yes_or_no[9]))
            draw_quads(p[4],p[10],ptBlur[10],ptBlur[4],1);
```

```
    if ((yes_or_no[9] || yes_or_no[10])
            && !(yes_or_no[9] && yes_or_no[10]))
                draw_quads(p[5],p[11],ptBlur[11],ptBlur[5],1);
    if ((yes_or_no[10] || yes_or_no[11])
            && !(yes_or_no[10] && yes_or_no[11]))
                draw_quads(p[6],p[12],ptBlur[12],ptBlur[6],1);
    if ((yes_or_no[11] || yes_or_no[6])
            && !(yes_or_no[11] && yes_or_no[6]))
                draw_quads(p[1],p[7],ptBlur[7],ptBlur[1],1);
    //neb
    if ((yes_or_no[0] || yes_or_no[1])
            && !(yes_or_no[0] && yes_or_no[1]))
                draw_quads(p[0],p[2],ptBlur[2],ptBlur[0],1);
    if ((yes_or_no[1] || yes_or_no[2])
            && !(yes_or_no[1] && yes_or_no[2]))
                draw_quads(p[0],p[3],ptBlur[3],ptBlur[0],1);
    if ((yes_or_no[2] || yes_or_no[3])
            && !(yes_or_no[2] && yes_or_no[3]))
                draw_quads(p[0],p[4],ptBlur[4],ptBlur[0],1);
    if ((yes_or_no[3] || yes_or_no[4])
            && !(yes_or_no[3] && yes_or_no[4]))
                draw_quads(p[0],p[5],ptBlur[5],ptBlur[0],1);
    if ((yes_or_no[4] || yes_or_no[5])
            && !(yes_or_no[4] && yes_or_no[5]))
                draw_quads(p[0],p[6],ptBlur[6],ptBlur[0],1);
    if ((yes_or_no[5] || yes_or_no[0])
            && !(yes_or_no[5] && yes_or_no[0]))
                draw_quads(p[0],p[1],ptBlur[1],ptBlur[0],1);
    glDisable(GL_BLEND);
}

//笔杆材料
void Pencil::material_holder()
{
    glColor4fv(holder_d.c);
    glMaterialfv(GL_FRONT_AND_BACK,GL_AMBIENT,holder_a.c);
    glMaterialfv(GL_FRONT_AND_BACK,GL_DIFFUSE,holder_d.c);
    glMaterialfv(GL_FRONT,GL_SPECULAR,holder_s.c);
    glMaterialf(GL_FRONT,GL_SHININESS,60);
}

//笔尖和笔帽材料
void Pencil::material_cap_neb()
{
```

```
        glColor4fv(neb_cap_d.c);
        glMaterialfv(GL_FRONT_AND_BACK,GL_AMBIENT,neb_cap_a.c);
        glMaterialfv(GL_FRONT_AND_BACK,GL_DIFFUSE,neb_cap_d.c);
        glMaterialfv(GL_FRONT,GL_SPECULAR,neb_cap_s.c);
        glMaterialf(GL_FRONT,GL_SHININESS,60);
}

//阴影的颜色
void Pencil::material_shadow()
{
        GLfloat mat_amb[] = {0.15f,0.15f,0.15f,1.0f};
        GLfloat mat_spe[] = {0.7f,0.7f,0.7f,1.0f};

        glColor4fv(shadow_c.c);
        glMaterialfv(GL_FRONT_AND_BACK,GL_AMBIENT,mat_amb);
        glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE,shadow_c.c);
        glMaterialfv(GL_FRONT,GL_SPECULAR,mat_spe);
        glMaterialf(GL_FRONT,GL_SHININESS,30);
}

//半影的颜色
void Pencil::material_penumbra()
{
        glColor4fv(penu_c.c);
        glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, penu_c.c);
}

void Pencil::set_holder_mat(GLfloat * a, GLfloat * d, GLfloat * s)
{
        holder_a = a;
        holder_d = d;
        holder_s = s;
}

//设置笔的材质
void Pencil::set_neb_cap_mat(GLfloat * a, GLfloat * d, GLfloat * s)
{
        neb_cap_a = a;
        neb_cap_d = d;
        neb_cap_s = s;
}

//设置阴影及半影颜色
void Pencil::set_shadow_penu_mat(GLfloat * shad_clr, GLfloat * penu_clr)
```

```
{
    shadow_c = shad_clr;
    penu_c = penu_clr;
}
```

ShadowView.h 文件：

```
////////////////////////////////////////////////////////////////////////////////
//////////////////////////                          //////////////////////////
//////////////////////////        ShadowView.h      //////////////////////////
//////////////////////////                          //////////////////////////
////////////////////////////////////////////////////////////////////////////////

// ShadowView.h : interface of the CShadowView class
//
////////////////////////////////////////////////////////////////////////////

#include "gl\gl.h"
#include "gl\glu.h"

#if !defined(AFX_SHADOWVIEW_H__2F29B08D_8086_11D2_83B3_0080C832FCF3__
        INCLUDED_)
#define AFX_SHADOWVIEW_H__2F29B08D_8086_11D2_83B3_0080C832FCF3__INCLUDED_

#if _MSC_VER >= 1000
#pragma once
#endif // _MSC_VER >= 1000

class CShadowView : public CView
{
protected: // create from serialization only
    CShadowView();
    DECLARE_DYNCREATE(CShadowView)

// Attributes
public:
    CShadowDoc* GetDocument();
    CClientDC    *m_pDC;
    CRect        m_oldRect;

// Operations
public:

// Overrides
    // ClassWizard generated virtual function overrides
```

```
        //{{AFX_VIRTUAL(CShadowView)
        public:
        virtual void OnDraw(CDC* pDC); // overridden to draw this view
        virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
        protected:
        //}}AFX_VIRTUAL

// Implementation
public:
        virtual ~CShadowView();
#ifdef _DEBUG
        virtual void AssertValid() const;
        virtual void Dump(CDumpContext& dc) const;
#endif

protected:
        //光的照射方向:(-m_lightPos_x, -m_lightPos_y, -m_lightPos_z)
        GLfloat         m_lightPos_x;
        GLfloat         m_lightPos_y;
        GLfloat         m_lightPos_z;
// Generated message map functions
protected:

        void Ground();
        void CreateSun();
        void DrawScene();
        BOOL bSetupPixelFormat();
        void Init();
        //{{AFX_MSG(CShadowView)
        afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
        afx_msg void OnDestroy();
        afx_msg void OnSize(UINT nType, int cx, int cy);
        //}}AFX_MSG
        DECLARE_MESSAGE_MAP()
};

#ifndef _DEBUG // debug version in ShadowView.cpp
inline CShadowDoc* CShadowView::GetDocument()
        { return (CShadowDoc*)m_pDocument; }
#endif

//{{AFX_INSERT_LOCATION}}
// Microsoft Developer Studio will insert additional declarations immediately before the previous //line.
#endif // !defined(AFX_SHADOWVIEW_H__2F29B08D_8086_11D2_83B3_0080C832FCF3__
```

```
INCLUDED_)
```

ShadowView.cpp 文件：

```
////////////////////////////////////////////////////////////////////////////////////////////
//////////////////////////                                  //////////////////////////
//////////////////////////          ShadowView.cpp          //////////////////////////
//////////////////////////                                  //////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////

// ShadowView.cpp : implementation of the CShadowView class

#include "stdafx.h"
#include "Shadow.h"

#include "ShadowDoc.h"
#include "ShadowView.h"

#include "math.h"
#include "point.h"
#include "pencil.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

////////////////////////////////////////////////////////////////////////////
// CShadowView

IMPLEMENT_DYNCREATE(CShadowView, CView)

BEGIN_MESSAGE_MAP(CShadowView, CView)
    //{{AFX_MSG_MAP(CShadowView)
    ON_WM_CREATE()
    ON_WM_DESTROY()
    ON_WM_SIZE()
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

////////////////////////////////////////////////////////////////////////////
// CShadowView construction/destruction
```

```
CShadowView::CShadowView()
{
	// TODO: add construction code here
	m_pDC = NULL;

	m_lightPos_x = 1.0f;
	m_lightPos_y = -1.0f;
	m_lightPos_z = 1.3f;
}

CShadowView::~CShadowView()
{
}

BOOL CShadowView::PreCreateWindow(CREATESTRUCT& cs)
{
	cs.style |= WS_CLIPCHILDREN | WS_CLIPSIBLINGS;
	return CView::PreCreateWindow(cs);
}

/////////////////////////////////////////////////////////////////////////////
// CShadowView drawing

void CShadowView::OnDraw(CDC* pDC)
{
	CShadowDoc* pDoc = GetDocument();
	ASSERT_VALID(pDoc);

	// TODO: add draw code for native data here
	DrawScene();
}

/////////////////////////////////////////////////////////////////////////////
// CShadowView diagnostics

#ifdef _DEBUG
void CShadowView::AssertValid() const
{
	CView::AssertValid();
}

void CShadowView::Dump(CDumpContext& dc) const
{
```

```
    CView::Dump(dc);
}

CShadowDoc * CShadowView::GetDocument() // non-debug version is inline
{
    ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(CShadowDoc)));
    return (CShadowDoc *)m_pDocument;
}
#endif //_DEBUG

/////////////////////////////////////////////////////////////////////////////
// CShadowView message handlers

void CShadowView::Init()
{
    PIXELFORMATDESCRIPTOR pfd;
    int         n;
    HGLRC       hrc;
    m_pDC = new CClientDC(this);
    ASSERT(m_pDC != NULL);
    if (!bSetupPixelFormat()) return;
    n = ::GetPixelFormat(m_pDC->GetSafeHdc());
    ::DescribePixelFormat(m_pDC->GetSafeHdc(), n, sizeof(pfd), &pfd);
    hrc = wglCreateContext(m_pDC->GetSafeHdc());
    wglMakeCurrent(m_pDC->GetSafeHdc(), hrc);
    GetClientRect(&m_oldRect);
    glClearDepth(1.0f);
    glEnable(GL_DEPTH_TEST);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

BOOL CShadowView::bSetupPixelFormat()
{
    static PIXELFORMATDESCRIPTOR pfd =
    {
      sizeof(PIXELFORMATDESCRIPTOR),
      1,
      PFD_DRAW_TO_WINDOW |
      PFD_SUPPORT_OPENGL,
      PFD_TYPE_RGBA,
      24,
      0, 0, 0, 0, 0, 0,
      0,
```

```
        0,
        0,
        0, 0, 0, 0,
        32,
        0,
        0,
        PFD_MAIN_PLANE,
        0,
        0, 0, 0
    };
    int pixelformat;
    if ( (pixelformat = ChoosePixelFormat(m_pDC->GetSafeHdc(), &pfd)) == 0 )
    {
        MessageBox("ChoosePixelFormat failed");
        return FALSE;
    }
    if (SetPixelFormat(m_pDC->GetSafeHdc(), pixelformat, &pfd) == FALSE)
    {
        MessageBox("SetPixelFormat failed");
        return FALSE;
    }
    return TRUE;
}

int CShadowView::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    if (CView::OnCreate(lpCreateStruct) == -1)
        return -1;
    Init();
    return 0;
}

void CShadowView::OnDestroy()
{
    HGLRC hrc;
    hrc = ::wglGetCurrentContext();
    ::wglMakeCurrent(NULL, NULL);
    if (hrc)
        ::wglDeleteContext(hrc);
    if (m_pDC)
        delete m_pDC;
    CView::OnDestroy();
}
```

```
void CShadowView::OnSize(UINT nType, int cx, int cy)
{
    CView::OnSize(nType, cx, cy);
    GLfloat aspect;
    glViewport( 0, 0, cx, cy);
    aspect = (GLfloat) cx / cy;
    glMatrixMode( GL_PROJECTION );
    glLoadIdentity();
    gluPerspective(45.0, aspect, 1.0, 10.0 );
}

void CShadowView::DrawScene()
{
    glClearColor(0.0f,0.0f,0.0f,1.0f);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glTranslatef(0.0f,0.0f,-4.5f);
    glFrontFace(GL_CCW);
    glTranslatef(0.0f,-1.2,0.0);

    //    将图形坐标系变成笛卡尔坐标系
    glRotatef(-90.0,1.0f,0.0f,0.0f);
    glRotatef(-90.0,0.0f,0.0f,1.0f);
    CreateSun();
    Point sun_direction;
    Pencil pencil;
    //设置光的照射方向
    sun_direction.set(-m_lightPos_x,-m_lightPos_y,-m_lightPos_z);
    //设置铅笔的尺寸
    pencil.SetPencil(0.08,0.56,2.0);
    pencil.translate_parameter(0.5,0.0,0.0);
    //    请自行作一些变换,例如:
    //    pencil.rotate_angle_x(75.0*3.14159265/180.0);
    //    pencil.translate_parameter(0.5,0.0,0.5);
    //绘制铅笔及其阴影
    pencil.draw();
    pencil.draw_shadow(sun_direction);
    //地板
    Ground();
    glFinish();
}

void CShadowView::CreateSun()
```

```
{
    GLfloat light0Amb[] = {0.0f,0.0f,0.0f,1.0f};
    GLfloat light0Dif[] = {0.9f,0.9f,0.9f,1.0f};
    GLfloat light0Spe[] = {1.0f,1.0f,1.0f,1.0f};

    GLfloat light0Pos[] = {m_lightPos_x,m_lightPos_y,m_lightPos_z,0.0f};

    //    第0盏灯为太阳,第1盏灯为聚光灯
    //    为了简化,本程序假定聚光不产生阴影:
    glLightfv(GL_LIGHT0,GL_AMBIENT,light0Amb);
    glLightfv(GL_LIGHT0,GL_DIFFUSE,light0Dif);
    glLightfv(GL_LIGHT0,GL_SPECULAR,light0Spe);
    glLightfv(GL_LIGHT0,GL_POSITION,light0Pos);
    GLfloat globalAmb[] = {0.2f,0.2f,0.2f,1.0f};
    glLightModelfv(GL_LIGHT_MODEL_AMBIENT,globalAmb);

    GLfloat spotAmb[] = {0.2f,0.2f,0.2f,0.8f};
    GLfloat spotDif[] = {0.7f,0.7f,0.7f,0.8f};
    GLfloat spotSpe[] = {0.7f,0.7f,0.7f,0.8f};
    GLfloat spotPos[] = {1.0f,0.8f,1.3f,0.8f};
    GLfloat spotDir[] = {-1.0f,-0.8f,-.80f,1.0f};
    GLfloat spotCutoff = 45.0f;
    glLightfv(GL_LIGHT1,GL_AMBIENT,spotAmb);
    glLightfv(GL_LIGHT1,GL_DIFFUSE,spotDif);
    glLightfv(GL_LIGHT1,GL_SPECULAR,spotSpe);
    glLightfv(GL_LIGHT1,GL_POSITION,spotPos);
    glLightfv(GL_LIGHT1,GL_SPOT_DIRECTION,spotDir);
    glLightf(GL_LIGHT1,GL_SPOT_CUTOFF,spotCutoff);
    glLightf(GL_LIGHT1,GL_CONSTANT_ATTENUATION,0.3);

    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    glEnable(GL_LIGHT1);
}

//创建地板
void CShadowView::Ground()
{
    glNormal3f(0.0f,0.0f,1.0f);
    glPushAttrib(GL_LIGHTING_BIT);
        glDisable(GL_LIGHTING);
        glBegin(GL_QUADS);
            glColor4f(0.0f,0.0f,0.0f,1.0f);
            glVertex3f(-5.0f,-2.0f,0.0f);
```

```
            glColor4f(0.70f,0.70f,0.70f,1.0f);
            glVertex3f(1.60f, -2.0f,0.0f);
            glColor4f(0.70f,0.70f,0.70f,1.0f);
            glVertex3f(1.60f,2.0f,0.0f);
            glColor4f(0.0f,0.0f,0.0f,1.0f);
            glVertex3f(-5.0f,2.0f,0.0f);
        glEnd();
    glPopAttrib();
}
```

16.6 牛头刨床急回机构的运动仿真

16.6.1 程序设计思想

本章将牛头刨床机构进行简化，研究图 16-9 所示机构的运动问题。该机构由 OA 杆、O_1B 杆、套筒 M 及销钉 A 构成。套筒 M 和 OA 杆用销钉 A 连接，O_1B 杆则穿过套筒 M。很显然，OA 杆的运动会带动套筒 M 的运动，而套筒 M 的运动又将带动 O_1B 杆的运动。现在的问题是：第一，当 OA 杆匀速转动时，套筒 M 和 O_1B 杆的运动将是怎样的；第二，当一个观察者平躺在 OA 杆，其眼睛始终位于 O 点，随 OA 杆运动，且一旦躺在 OA 杆上之后，观察者自己不作任何位置的挪动，那么，观察者看到的 OA 杆、O_1B 杆和套筒 M 将作什么样的运动，也就是说，相对运动是什么。从运动分析的观点来讲，也就是以 O 为坐标原点，始终以 OA 杆为 X 轴建立一个随着 OA 杆转动的坐标系（如图 16-10），考察各构件相对于该系的运动。

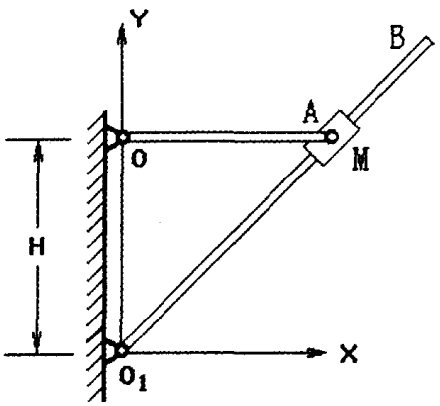

图 16-9 机构简图

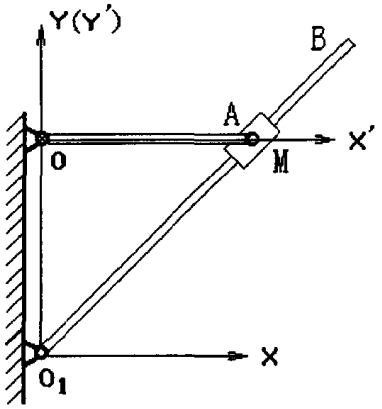

图 16-10 相对运动分析图

事实上，这是一类普通的运动仿真问题。下面，我们就来介绍完成这类问题的一般思路。

参数化三维图像的建立

建立可以描述问题的三维图像是应用 OpenGL 仿真的对象基础，问题化为 OpenGL 处理的对象即场景。场景主要由点、线、面等基元和图形基元组成。复杂的场景还需要良好的曲面生成算法。建立了场景的几何形体后，给各个对象或对象内部的基元进行材质定义、光照、线

或多边形反走样、光顺、透明与融合处理、纹理映射、雾化等一系列图像处理，以实现所期望的真实感。

实现图形的交互性，需进行参数化建模。在参数化的三维图形建模中可设定几个或几组控制参数，对几何模型进行分析，确定控制其几何形体的最佳效果，分析参数间的相关性、各对象对参数的依赖程度以及对象间的协调性。这些参数都是人机交互的接口。应用 MFC 编程环境是很容易做到这一点的。

运动参数的建立

运动参数用来确定运动对象在空间的几何位置。在 OpenGL 中这些参数是基于对象坐标和对象的局部坐标的。运动参数的多少一般就是运动系统的自由度数。

运动参数也可通过人机交互的接口，控制运动的速度。

运动的实现

绝对运动的实现，首先应根据给定的几何模型建立合理的数学模型。在实现实时仿真的过程中，结合运动对象在某瞬时的运动参数的数值，确定其空间的位置。OpenGL 实现的过程是：

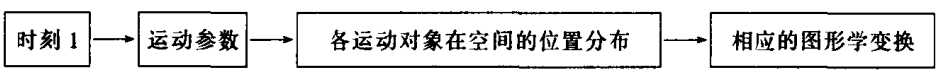

图 16-11　绝对运动的 OpenGL 实现

相对运动的实现是一件较困难的事。实现的思想就是把“相对运动是各运动对象相对于动参考系的运动”转化为 OpenGL 的表达方式，从而形成两条途径：一是直接采用相对坐标的描述实现其相对运动；二是基于已有的绝对运动实现其相对运动。第二种途径对于描述复杂系统的相对运动可起到事半功倍的效果，下面是 OpenGL 实现其相对运动的过程：

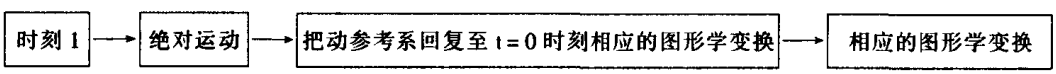

图 16-12　相对运动的 OpenGL 实现

这样，与动参考系回复至 t = 0 时刻的控制参数相应的图形学变换就从最高层控制着所有运动对象在空间的位置分布，从而轻松地实现相对运动。这种思想与 3D Studio R 系列、3D Studio MAX 版本的关节动画的链接思想是一致的。

运动观察

有时我们感兴趣的是局部的或某个运动对象的运动情况，从而需要从不同的角度去观察它们的绝对运动或相对运动、运动轨迹、速度和加速度等。为实现各种视窗效果，可设置 3 个参数，即沿 x、y、z 轴的转角来控制；对于平移和缩放可通过改变对象的位置坐标和比例变换来实现；对视口进行重新定位及调整，也可通过设置参数加以控制。所有这些参数均是人机交互的接口。

人机交互的实现

基于 OpenGL，运动仿真的人机交互包括如下几个方面：模型定制、动静控制、速度控制、内容控制、视角控制（摄像机定标）、场景变换控制（选择场景）。

对于一般性的软件，人机交互手段主要是通过键盘、鼠标等常用外设来实现的。对于高级

的实时仿真，不仅需要高性能的仿真设备，还需要高质量的仿真技术。比如头盔应用于 VR 技术。

画面艺术效果处理

仅仅演示了运动是不够的，还需要对画面进行艺术效果的处理。把美学、色彩学、人机工程学应用于 VC 程序设计中去，给画面增加新奇感、悦目感，并能培养兴趣，激发想象力、创造力。

综上所述，应用 OpenGL 实现机构运动实时仿真的总体思想可表示为图 16-13。

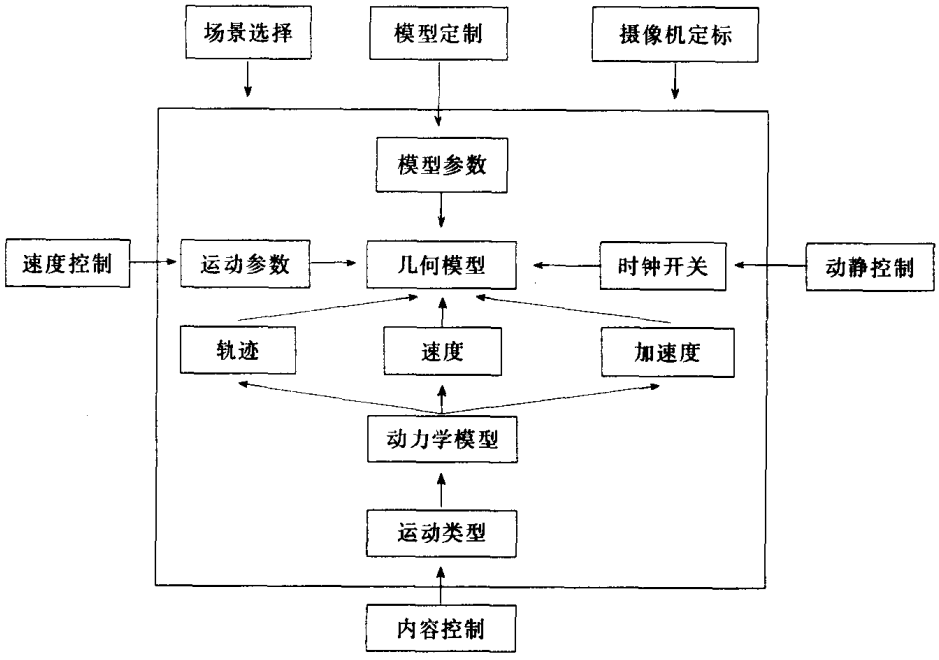

图 16-13　OpenGL 实现运动仿真

16.6.2　运动仿真的实现

利用 OpenGL 展示机构的绝对运动、相对运动以及相应的相对轨迹的形成过程，不仅具有良好的交互性，而且也具有启发性，对难以想象的相对运动进行了真实的模拟。现在来完成前面所提的两个问题。

下面我们所要模拟的牛头刨床急回机构的运动是从刚体运动学的观点出发来研究的，不涉及动力学和运动特性分析。

本程序的人机交互的接口是通过以下几方面的控制参数来实现的。

模型参数：铰曲面参数、铰厚度参数、杆半径参数、两杆长度参数、两铰距离参数；

运动参数：OA 杆与 x 轴的夹角；

动静参数：运动与静止开关；

内容参数：绝对运动、相对运动、随绝对运动展示的相对运动及其运动轨迹类型参数。

(1) 建立工程

建立一个名为 Jgfz 的工程，并按第 5 章中的方法极小化 OpenGL。最后参考一下程序清单，看看是否有不同之处。

(2) 按 JgfzView.h、JgfzView.cpp 文件的程序清单添加枚举变量、保护成员变量、保护成员函数,增加宏定义和文件包含。

(3) 添加菜单并响应菜单

添加一个 Caption 名为 C&ontrol 的菜单。加入菜单项。其 Caption 和 ID 号如表 16-4 所示。再添加一个 Caption 名为 Spee&d 的菜单,加入菜单项,其 Caption 和 ID 号如表 16-5。

表 16-4

Caption	I D
S&tatic	IDM _ CONTROL _ STATIC
&Move	IDM _ CONTROL _ MOVE
分隔符	
&Absolute Movement	IDM _ ABSOLUTEMOVEMENT
分隔符	
&Relative Trace 1	IDM _ TRACE _ 1
R&elative Move 1	IDM _ RELATIVEMOVE _ 1

表 16-5

Caption	I D
S&low	IDM _ SPEED _ SLOW
&Fast	IDM _ SPEED _ FAST
&Ultra	IDM _ SPEED _ ULTRA

请按程序清单进行响应。

(4) 为工程添加一个 C 语言头文件 my.h,并按程序列表进行编辑。

(5) 包含 glaux.lib 函数库文件。

(6) 运行程序

编译连接并运行程序,执行菜单 Control \ Move,结果机构开始运动,如图 16-14 所示;执行 Control \ Static,机构停止运动;执行菜单 Control \ Relative Trace 1,可以看到相对运动轨迹的形成过程,如图 16-15;执行菜单 Control \ R&elative Move 1,机构各部分表现的运动就是当初的观察者实际所看到的运动(这种运动若是仅仅在纸上分析能较直观地总结出来吗?),如图 16-16。还可以执行速度控制参数,对机构运动的速度加以选择。

本程序实时地模拟了工程中牛头刨床急回机构的运动,研究的方法源于刚体运动学,因而最直观的反映了运动对象的运动。借助于一个点的运动,从而分析了整个机构的运动。通常还用其他两种方法分析该机构的运动,其一是动点仍然选择初始位置时 O_1B 杆上与销钉 A 点相重合的点,而动参考系则选择随着 OA 杆一起平动,其坐标原点为 A,即在运动的过程中,动参考系始终保持方位不变;其二是动点选择销钉 A 点,动参考系则固结在 O_1B 杆上,其坐标原点为 O_1 点,x 轴沿 O_1B。基于这两种研究对象和运动分析,各运动对象的绝对运动不会发生改变,但相对运动却有很大的不同。由于研究方法和编程技术相同,加之篇幅因素,我们在此不进一步分析。

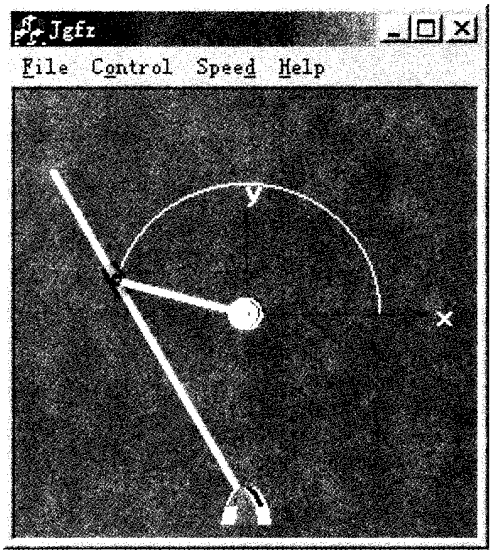

图 16-14　机构的运动

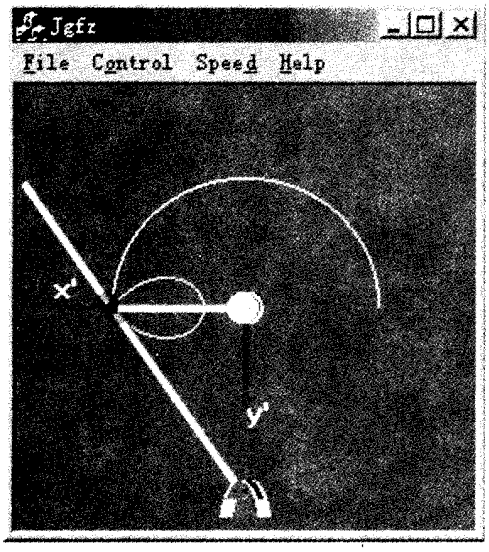

图 16-15　相对运动的分析

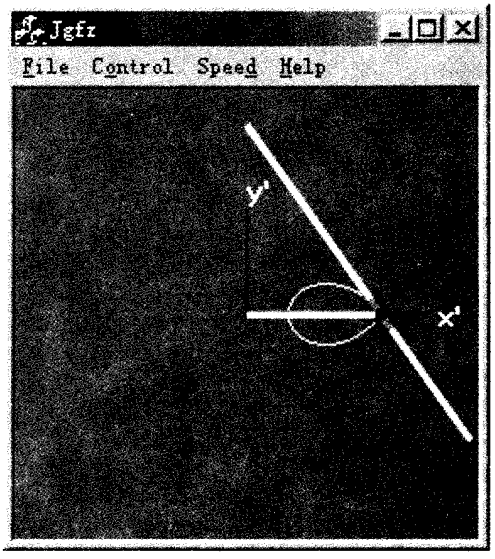

图 16-16　相对运动

16.6.3　程序清单

JgfzView.h 文件：

```
//////////////////////////////////////////////////////////////////////////////////////////
//////////////////////////                                  //////////////////////////
//////////////////////////          JgfzView.h              //////////////////////////
//////////////////////////                                  //////////////////////////
```

```
////////////////////////////////////////////////////////////////////////////////

// JgfzView.h : interface of the CJgfzView class

#include "gl\gl.h"
#include "gl\glu.h"

#if !defined(AFX_JGFZVIEW_H__AD0CFFCD_8176_11D2_83B3_0080C832FCF3__INCLUDED_)
#define AFX_JGFZVIEW_H__AD0CFFCD_8176_11D2_83B3_0080C832FCF3__INCLUDED_

#if _MSC_VER >= 1000
#pragma once
#endif // _MSC_VER >= 1000

#define    BAR_1                    1
#define    BAR_2                    2
#define    TAOTONG                  3
#define    TAOGUA                   4
#define    ZOUCHENG_X               5
#define    ZIZUOUP_1                6
#define    ZIZUOUP_2                7
#define    ZOUCHENG_S               8

enum{Move, Static};
enum{AbsoluteMovement,RelativeTrace1,RelativeMovement1,RelativeTrace2,
                RelativeMovement2,RelativeTrace3,RelativeMovement3};

class CJgfzView : public CView
{
protected: // create from serialization only
    CJgfzView();
    DECLARE_DYNCREATE(CJgfzView)

// Attributes
public:
    CJgfzDoc* GetDocument();
    CClientDC   *m_pDC;
    CRect       m_oldRect;

// Operations
public:

// Overrides
    // ClassWizard generated virtual function overrides
```

```
        //{{AFX_VIRTUAL(CJgfzView)
        public:
        virtual void OnDraw(CDC* pDC); // overridden to draw this view
        virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
        protected:
        //}}AFX_VIRTUAL

// Implementation
public:
        virtual ~CJgfzView();
#ifdef _DEBUG
        virtual void AssertValid() const;
        virtual void Dump(CDumpContext& dc) const;
#endif

protected:
        UINT            m_MoveStatic;

        GLfloat         m_rotateAngle;
        GLfloat         m_speedFactor;

        GLfloat         m_height;
        GLfloat         m_bar1Length;
        GLfloat         m_bar2Length;
        GLfloat         m_radius;
        GLfloat         m_taotongLength;
        GLfloat         m_taotongRadius;
        BOOL            m_radiusFlag;

        GLdouble        m_zizuoCoefficient;
        GLdouble        m_zizuoThick;
        GLdouble        m_zizuoWidth1;
        GLdouble        m_zizuoWidth2;
        GLdouble        m_zizuoOffset;

        UINT            m_absoluteRelative;
        BOOL            m_copyImage;

// Generated message map functions
protected:
        void EnableLighting();
        void DrawRelativeCurve1();
        void DrawBackground();
        void DrawXY();
```

```
    void DrawCoordinate();
    void DrawStatic();
    void DrawMove();
    void DrawScene();
    BOOL bSetupPixelFormat();
    void Init();
    //{{AFX_MSG(CJgfzView)
    afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
    afx_msg void OnDestroy();
    afx_msg void OnSize(UINT nType, int cx, int cy);
    afx_msg void OnTimer(UINT nIDEvent);
    afx_msg void OnAbsolutemovement();
    afx_msg void OnControlMove();
    afx_msg void OnControlStatic();
    afx_msg void OnRelativemove1();
    afx_msg void OnTrace1();
    afx_msg void OnSpeedFast();
    afx_msg void OnSpeedSlow();
    afx_msg void OnSpeedUltra();
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};

#ifndef _DEBUG // debug version in JgfzView.cpp
inline CJgfzDoc* CJgfzView::GetDocument()
    { return (CJgfzDoc*)m_pDocument; }
#endif

/////////////////////////////////////////////////////////////////////////////

//{{AFX_INSERT_LOCATION}}
// Microsoft Developer Studio will insert additional declarations immediately before the previous line.

#endif // !defined(AFX_JGFZVIEW_H__AD0CFFCD_8176_11D2_83B3_0080C832FCF3__
                INCLUDED_)
```

JgfzView.cpp 文件：

```
////////////////////////////////////////////////////////////////////////////////////////////
//////////////////////////                                      //////////////////////////
//////////////////////////          JgfzView.cpp                //////////////////////////
//////////////////////////                                      //////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////
```

```
// JgfzView.cpp : implementation of the CJgfzView class

#include "stdafx.h"
#include "Jgfz.h"

#include "JgfzDoc.h"
#include "JgfzView.h"

#include <math.h>
#include "my.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
// CJgfzView

IMPLEMENT_DYNCREATE(CJgfzView, CView)

BEGIN_MESSAGE_MAP(CJgfzView, CView)
    //{{AFX_MSG_MAP(CJgfzView)
    ON_WM_CREATE()
    ON_WM_DESTROY()
    ON_WM_SIZE()
    ON_WM_TIMER()
    ON_COMMAND(IDM_ABSOLUTEMOVEMENT, OnAbsolutemovement)
    ON_COMMAND(IDM_CONTROL_MOVE, OnControlMove)
    ON_COMMAND(IDM_CONTROL_STATIC, OnControlStatic)
    ON_COMMAND(IDM_RELATIVEMOVE_1, OnRelativemove1)
    ON_COMMAND(IDM_TRACE_1, OnTrace1)
    ON_COMMAND(IDM_SPEED_FAST, OnSpeedFast)
    ON_COMMAND(IDM_SPEED_SLOW, OnSpeedSlow)
    ON_COMMAND(IDM_SPEED_ULTRA, OnSpeedUltra)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CJgfzView construction/destruction

CJgfzView::CJgfzView()
{
```

```
    m_pDC = NULL;
    m_rotateAngle = 0.0f;                    // 角度自由度
    m_speedFactor = 15.0f;                   // 速度选择器
    m_MoveStatic = Static;                   // 运动与静止

    m_height = 0.84f;                        // 两铰之间的距离
    m_bar1Length = 0.56f;                    // 杆 1 的长度
    m_bar2Length = 1.7f;                     // 杆 2 的长度
    m_radius = 0.019f;                       // 杆的半径
    m_taotongLength = 0.16f;                 // 套筒长度
    m_taotongRadius = 0.04f;                 // 套筒半径
    m_radiusFlag = TRUE;                     // 系统模型自检标志

    m_zizuoCoefficient = 20;                 // 标准抛物线方程系数
    m_zizuoThick = 0.3;                      // 支座长度
    m_zizuoWidth1 = 0.13;                    // 支座宽度
    m_zizuoWidth2 = 0.16;                    // 支座宽度
    m_zizuoOffset = 0.5;                     // 偏移量,决定固定铰的开度

    m_absoluteRelative = AbsoluteMovement; // 运动类型控制参数
    m_copyImage = TRUE;                      // 图像输出控制
}

CJgfzView::~CJgfzView()
{
}

BOOL CJgfzView::PreCreateWindow(CREATESTRUCT& cs)
{
    cs.style |= WS_CLIPCHILDREN | WS_CLIPSIBLINGS;
    return CView::PreCreateWindow(cs);
}

void CJgfzView::OnDraw(CDC* pDC)
{
    CJgfzDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    DrawScene();
}

#ifdef _DEBUG
void CJgfzView::AssertValid() const
{
    CView::AssertValid();
```

```
}

void CJgfzView::Dump(CDumpContext& dc) const
{
    CView::Dump(dc);
}

CJgfzDoc* CJgfzView::GetDocument() // non-debug version is inline
{
    ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(CJgfzDoc)));
    return (CJgfzDoc*)m_pDocument;
}
#endif //_DEBUG

void CJgfzView::Init()
{
    PIXELFORMATDESCRIPTOR pfd;
    int         n;
    HGLRC       hrc;
    m_pDC = new CClientDC(this);
    ASSERT(m_pDC != NULL);
    if (!bSetupPixelFormat())       return;
    n = ::GetPixelFormat(m_pDC->GetSafeHdc());
    ::DescribePixelFormat(m_pDC->GetSafeHdc(), n, sizeof(pfd), &pfd);
    hrc = wglCreateContext(m_pDC->GetSafeHdc());
    wglMakeCurrent(m_pDC->GetSafeHdc(), hrc);
    GetClientRect(&m_oldRect);
    glClearDepth(1.0f);
    glEnable(GL_DEPTH_TEST);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

BOOL CJgfzView::bSetupPixelFormat()
{
    static PIXELFORMATDESCRIPTOR pfd =
    {
      sizeof(PIXELFORMATDESCRIPTOR),
      1,
      PFD_DRAW_TO_WINDOW | PFD_SUPPORT_OPENGL |
            PFD_DOUBLEBUFFER,
      PFD_TYPE_RGBA,
      24,
      0, 0, 0, 0, 0, 0,
```

```
        0,
        0,
        0,
        0, 0, 0, 0,
        32,
        0,
        0,
        PFD_MAIN_PLANE,
        0,
        0, 0, 0
    };
    int pixelformat;
    if ( (pixelformat = ChoosePixelFormat(m_pDC->GetSafeHdc(), &pfd)) == 0 )
    {
        MessageBox("ChoosePixelFormat failed");
        return FALSE;
    }
    if (SetPixelFormat(m_pDC->GetSafeHdc(), pixelformat, &pfd) == FALSE)
    {
        MessageBox("SetPixelFormat failed");
        return FALSE;
    }
    return TRUE;
}

int CJgfzView::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    if (CView::OnCreate(lpCreateStruct) == -1)
        return -1;
    Init();
    return 0;
}

void CJgfzView::OnDestroy()
{
    HGLRC hrc;
    hrc = ::wglGetCurrentContext();
    ::wglMakeCurrent(NULL, NULL);
    KillTimer(1);
    if (hrc)
        ::wglDeleteContext(hrc);
    if (m_pDC)
        delete m_pDC;
    CView::OnDestroy();
```

```
}

void CJgfzView::OnSize(UINT nType, int cx, int cy)
{
    CView::OnSize(nType, cx, cy);
    if(cy > 0)
    {
        if((m_oldRect.right > cx) || (m_oldRect.bottom > cy))
        RedrawWindow();
      m_oldRect.right = cx;
      m_oldRect.bottom = cy;
      glMatrixMode(GL_PROJECTION);
      glLoadIdentity();
        EnableLighting();
        glFrustum(-1.0,1.0,-1.0,1.0,0.0,7.0);
        glViewport(0, 0, cx, cy);
    }
}

void CJgfzView::DrawScene()
{
    glShadeModel(GL_SMOOTH);
    glEnable(GL_DEPTH_TEST);
    glClearColor(0.1f,0.4f,0.5f,1.0f);
    glClear(GL_COLOR_BUFFER_BIT |GL_DEPTH_BUFFER_BIT);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glTranslatef(0.0f,0.0f,-0.5f);
    //背景:
    glPushMatrix();
        DrawBackground();
    glPopMatrix();
    glPushMatrix();
        DrawMove();
    glPopMatrix();

    // 如果显示相对运动,就不画两个支座:
    glPushMatrix();
        if((m_absoluteRelative==AbsoluteMovement) |
            (m_absoluteRelative==RelativeTrace1))
            DrawStatic();
    glPopMatrix();
    glFinish();
    SwapBuffers(wglGetCurrentDC());
```

```
}

//绘制机构的运动对象,完成绝对运动和相对运动的实时再现模拟
//OA 杆为第一杆,O1B 杆为第二杆
void CJgfzView::DrawMove()
{
    LENGTHANGLEINFO lengthAngleInfo;    //保存第 2 杆的运动信息
    switch(m_absoluteRelative)
    {
        case RelativeTrace1 :          //在地面上观看相对运动的形成过程
            glPushMatrix();
                glRotatef(m_rotateAngle,0.0f,0.0f,1.0f);
                DrawXY();
                //    画相对轨迹:
                DrawRelativeCurve1();
            glPopMatrix();
            break;
        case RelativeMovement1 :      //在 OA 杆上所看到的整个机构运动对象的运动
            glPushMatrix();
                DrawXY();
                //    画相对轨迹:
                DrawRelativeCurve1();
            glPopMatrix();
            break;
        default:    break;
    }
    //    完成平面转换,从 Oxy 到 Ozy:
    glRotatef(90.0f,0.0f,1.0f,0.0f);
    //    当显示相对运动时,就不画绝对运动的轨迹:
    if((m_absoluteRelative==AbsoluteMovement)||
            (m_absoluteRelative==RelativeTrace1))
        DrawCircleRoad(m_bar1Length,m_radius,m_rotateAngle,1.0f);
    //    进行融合处理
    glEnable(GL_BLEND);
    glEnable(GL_DEPTH_TEST);
    glBlendFunc(GL_ONE,GL_ZERO);
    //    实现相对运动:
    switch(m_absoluteRelative)
    {
        case RelativeMovement1 :
            glRotatef(m_rotateAngle,1.0f,0.0f,0.0f);
            break;
        default :    break;
    }
```

```
//        绝对运动思想
//            绘制第一根杆及其运动
glPushMatrix();
    //    已经重新设置过系统的坐标中心：移位,向前
    glTranslatef(m_taotongRadius + m_radius + 0.01f,0.0f,0.0f);
    //开始实现机构的绝对运动
    glRotatef(-m_rotateAngle,1.0f,0.0f,0.0f);
    //着色挂钩：
    DrawTaogua(m_bar1Length,m_radius);

    GLfloat materialAD[] = {1.0f,1.0f,1.0f,0.0f};
    glMaterialfv(GL_FRONT,GL_AMBIENT_AND_DIFFUSE,materialAD);
    //    着色杆 1
    DrawBar(BAR_1,m_radius,m_bar1Length);
    glCallList(BAR_1);
glPopMatrix();
//    绘制第二根杆及其运动
glPushMatrix();
    //    已经重新设置过系统的坐标中心
    if(m_absoluteRelative! = RelativeMovement3)
            glTranslatef(0.0f,-m_height,0.0f);
    //    获取杆 2 及套筒的运动信息
    if(m_absoluteRelative ! = RelativeTrace3)
        lengthAngleInfo = ClacuAngleBar2(m_bar1Length,
            m_radius,m_height,m_rotateAngle);
    glRotated(-lengthAngleInfo.bar2RotateAngle,1.0f,0.0f,0.0f);

    GLfloat materialAD2[] = {1.0f,1.0f,1.0f,0.0f};
    glMaterialfv(GL_FRONT,GL_AMBIENT_AND_DIFFUSE,materialAD2);
    DrawBar(BAR_2,m_radius,m_bar2Length);
    glCallList(BAR_2);
    //    套筒:进行融合处理
    glDepthMask(FALSE);
    glBlendFunc(GL_SRC_ALPHA,GL_ONE_MINUS_SRC_ALPHA);

    glPushMatrix();
        //    动画修改区域,套筒的运动：
        glTranslated(0.0,0.0,
                lengthAngleInfo.taotongTranslate - m_taotongLength/2.0);
        GLfloat materialAD3[] = {0.10f,0.5f,0.0f,0.9f};
        GLfloat materialSP[] = {0.5f,0.5f,0.5f,0.8f};
        glMaterialfv(GL_FRONT,GL_SPECULAR,materialSP);
        glMaterialf(GL_FRONT,GL_SHININESS,50.0);
```

```
            glMaterialfv(GL_FRONT,GL_AMBIENT_AND_DIFFUSE,materialAD3);
            DrawBar(TAOTONG, m_taotongRadius,m_taotongLength);
            glCallList(TAOTONG);
        glPopMatrix();

        //    显示动点:
        if(   (m_absoluteRelative = = RelativeTrace1) ||
            (m_absoluteRelative = = RelativeMovement1))
        {
            GLfloat ps[2];
            glGetFloatv(GL_POINT_SIZE_RANGE,ps);
            glPointSize(ps[1]/2.0f);
            GLfloat zTranslate = (m_bar1Length + m_radius) * (m_bar1Length + m_radius)
                        + m_height * m_height;
            zTranslate = sqrt(zTranslate);
            glTranslatef(0.0f,0.0f,zTranslate);
            glBegin(GL_POINTS);
                glVertex3f(m_radius + 0.09f,0.0f,0.0f);
            glEnd();
        }
    glPopMatrix();

    //选择处理
    switch(m_absoluteRelative)
    {
        case RelativeTrace1:
        case RelativeMovement1 :
            //          如果不是绝对运动,则着色 x,y 轴,
            //          并进行内部反走样处理:
            glPushMatrix();
                //向前移位,以使坐标轴通过杆 1 的中心线
                glTranslatef(m_taotongRadius + m_radius + 0.01f,
                        0.0f,0.0f);
                //    如果不是绝对运动,则相对坐标应该运动,
                glRotatef(-m_rotateAngle,1.0f,0.0f,0.0f);
                //    显示相对坐标轴
                DrawCoordinate();
            glPopMatrix();
            break;
        default : break;
    }
    glDepthMask(GL_TRUE);
    glDisable(GL_BLEND);
}
```

```
/*绘制机构的静止对象:上下两个支座,上支座由几个厚度不大的圆柱组成,而下支座则是由一个位于o_xy
平面上的曲线(由一段抛物线和两段直线组成)的拉伸所构成的一个三维形体和拷贝该三维形体然后在z轴
方向移动组成两片,由于两片在空间上有一定的距离,因而在它们的底部用两个平面来连接,这就像是把一个
较大的片从中间挖一个槽。在两片的上部穿过一个销钉,它用一个圆柱来代替,O1B杆就是绕该轴转动的*/
void CJgfzView::DrawStatic()
{
    GLdouble yOffset;
    GLdouble zOffset;
    yOffset = m_height - m_zizuoCoefficient
        * m_zizuoWidth1 * m_zizuoWidth1/4.0
        + m_radius;
    zOffset = m_zizuoThick + m_radius + m_zizuoOffset/2.0;

    //    画绝对坐标系,其颜色将会受到融合的影响:
    if(m_absoluteRelative == AbsoluteMovement)
    {
        glPushMatrix();
            DrawXY();
            //    因为坐标系位于Oyz平面
            glRotatef(90.0f,0.0f,1.0f,0.0f);
            DrawCoordinate();
        glPopMatrix();
    }
    //    画下支座:
    glPushMatrix();
        glRotatef(3.0f,0.0f,1.0f,0.0f);
        glTranslated(0.0f, -yOffset, -zOffset);
        DrawZiZuoXia(m_zizuoCoefficient,m_zizuoThick,
            m_zizuoWidth1,m_zizuoWidth2,m_zizuoOffset,
            m_radius);
        //    画下支座轴承:
        yOffset = -m_zizuoCoefficient * m_zizuoWidth1
            * m_zizuoWidth1/4.0 + m_radius;
        //两端加5倍
        glTranslated(0.0,yOffset, -5 * m_radius);
        GLfloat materialAD3[] = {1.0f,1.0f,1.0f,1.0f};
        GLfloat materialSP[] = {1.0f,1.0f,1.0f,1.0f};
        glMaterialfv(GL_FRONT,GL_SPECULAR,materialSP);
        glMaterialf(GL_FRONT,GL_SHININESS,100.0);
        glMaterialfv(GL_FRONT,GL_AMBIENT_AND_DIFFUSE,materialAD3);
        DrawBar(ZOUCHENG_X,m_radius + 0.006f,2.0f * (zOffset + 5 * m_radius));
        glCallList(ZOUCHENG_X);
```

```
        //    表面处理,(代码可以进一步向 DrawBar() 优化):
        GLfloat materialAD1[] = {1.0f,1.0f,1.0f,1.0f};
        GLfloat materialSP1[] = {1.0f,1.0f,1.0f,1.0f};
        glMaterialfv(GL_FRONT,GL_SPECULAR,materialSP1);
        glMaterialf(GL_FRONT,GL_SHININESS,100.0);
        glMaterialfv(GL_FRONT,GL_AMBIENT_AND_DIFFUSE,materialAD1);

        glTranslated(0.0,0.0,2.0*(zOffset+5*m_radius));
        GLUquadricObj* quadObj;
        quadObj = gluNewQuadric();
        gluQuadricDrawStyle(quadObj,GLU_FILL);
        gluQuadricOrientation(quadObj,GLU_OUTSIDE);
        gluDisk(quadObj,0.0,m_radius+0.006,10,1);
    glPopMatrix();

    //    画上支座:
    glPushMatrix();
        glRotatef(3.0f,0.0f,1.0f,0.0f);
        //    画上支座底盘(已经包含表面处理):
        GLfloat materialAD5[] = {0.7f,0.7f,0.7f,1.0f};
        GLfloat materialSP5[] = {1.0f,1.0f,1.0f,1.0f};
        glMaterialfv(GL_FRONT,GL_SPECULAR,materialSP5);
        glMaterialf(GL_FRONT,GL_SHININESS,100.0);
        glMaterialfv(GL_FRONT,GL_AMBIENT_AND_DIFFUSE,
            materialAD5);
        DrawBar(ZIZUOUP_1,4*m_radius,10*m_radius);
        glCallList(ZIZUOUP_1);
        //    画上支座顶盘(已经包含表面处理):
        glPushMatrix();
            glTranslatef(0.0f,0.0f,-14*m_radius);
            DrawBar(ZIZUOUP_2,3.5*m_radius,8*m_radius);
            glCallList(ZIZUOUP_2);
        glPopMatrix();
        //    画上支座轴承(已经包含表面处理):
        GLfloat materialAD6[] = {0.9f,0.9f,0.9f,1.0f};
        GLfloat materialSP6[] = {1.0f,1.0f,1.0f,1.0f};
        glMaterialfv(GL_FRONT,GL_SPECULAR,materialSP6);
        glMaterialf(GL_FRONT,GL_SHININESS,100.0);
        glMaterialfv(GL_FRONT,GL_AMBIENT_AND_DIFFUSE,
            materialAD6);

        glPushMatrix();
            glTranslatef(0.0f,0.0f,-20*m_radius);
            DrawBar(ZOUCHENG_S,1.5*m_radius,12*m_radius);
```

```
                glCallList(ZOUCHENG_S);
            glPopMatrix();
        glPopMatrix();
}

//绘制坐标系
void CJgfzView::DrawCoordinate()
{
        GLfloat xLength = m_barlLength + 12 * m_radius;
        if(m_absoluteRelative = = RelativeTrace2)
            xLength = m_barlLength;
        GLfloat yLength = 25 * m_radius;
        GLfloat lw[2];
        glGetFloatv(GL_LINE_WIDTH_RANGE,lw);
        glLineWidth(lw[1]/12.0);
        // 如果只显示绝对运动坐标系,就没有必要进行反走样:
        if(m_absoluteRelative! = AbsoluteMovement)
        {
            glShadeModel(GL_FLAT);
            glEnable(GL_LINE_SMOOTH);
            glEnable(GL_BLEND);
            glBlendFunc(GL_SRC_ALPHA,GL_ONE_MINUS_SRC_ALPHA);
            glHint(GL_LINE_SMOOTH_HINT,GL_DONT_CARE);
        }
        glDisable(GL_LIGHTING);
        glColor3f(0.0f,0.0f,1.0f);
        //位于 Ozy 坐标:
        glBegin(GL_LINES);
            glVertex3f(0.0f,0.0f,0.0f);
            glVertex3f(0.0f,0.0f,xLength);

            glVertex3f(0.0f,0.0f,0.0f);
            glVertex3f(0.0f,yLength,0.0f);
        glEnd();
        //坐标箭头:
        glLineWidth(lw[1]/12.0);
        //Y 方向箭头:
        glPushMatrix();
            glTranslatef(0.0f,yLength,0.0f);
            glBegin(GL_LINES);
                glVertex3f(0.0f,0.0f,0.0f);
                glVertex3f(0.0f, -0.065f,0.01f);
                glVertex3f(0.0f,0.0f,0.0f);
                glVertex3f(0.0f, -0.065f, -0.01f);
```

```
        glEnd();
    glPopMatrix();

    //    Z方向箭头,但事实上显示为X方向。
    glPushMatrix();
        glTranslatef(0.0f,0.0f,xLength);
        glBegin(GL_LINES);
            glVertex3f(0.0f,0.0f,0.0f);
            glVertex3f(0.0f,0.01f,-0.065f);
            glVertex3f(0.0f,0.0f,0.0f);
            glVertex3f(0.0f,-0.01f,-0.065f);
        glEnd();
    glPopMatrix();
    glEnable(GL_LIGHTING);
    //    在绝对运动的情况下,没有进行坐标轴反走样:
    if(m_absoluteRelative! = AbsoluteMovement)
    {
        glDisable(GL_LINE_SMOOTH);
        glDisable(GL_BLEND);
    }
}

//打印XY等字符
void CJgfzView::DrawXY()
{
    //X',Y'字符位于坐标原点:
    CClientDC dc(this);
    SelectObject(dc.m_hDC,GetStockObject(SYSTEM_FONT));
    glDisable(GL_LIGHTING);
    glColor3f(1.0f,1.0f,1.0f);
    glPushMatrix();
        if(m_absoluteRelative= = RelativeTrace2)
            glTranslatef(m_barlLength+1.5*m_radius +0.015f,0.0f,0.0f);
        else
            glTranslatef(m_barlLength+13.5*m_radius +0.015f,-0.05f,0.0f);
        int iListGen=glGenLists(6);
        wglUseFontBitmaps(dc.m_hDC,0,256,1000);
        glRasterPos3f(0.0F,0.0F,0.0F);
        glListBase(1000);
        if(m_absoluteRelative= = AbsoluteMovement)
            glCallLists(1,GL_UNSIGNED_BYTE,"x");
        else
            glCallLists(2,GL_UNSIGNED_BYTE,"x'");
    glPopMatrix();
```

```
        glPushMatrix();
            glTranslatef(0.0f,26.5 * m_radius +0.015f,0.0f);
            wglUseFontBitmaps(dc.m_hDC,0,256,iListGen+3);
            glRasterPos3f(0.0F,0.0F,0.0F);
            glListBase(iListGen+3);
            if(m_absoluteRelative = = AbsoluteMovement)
                glCallLists(1,GL_UNSIGNED_BYTE,"y");
            else
                glCallLists(2,GL_UNSIGNED_BYTE,"y'");
        glPopMatrix();
        glEnable(GL_LIGHTING);
}

void CJgfzView::DrawBackground()
{
        //    画面的艺术处理:
}

//绘制相对运动的轨迹
void CJgfzView::DrawRelativeCurve1()
{
        VERTEXCOORDINATE coord;
        GLdouble initLength = sqrt((m_bar1Length + m_radius)
            * (m_bar1Length + m_radius) + m_height * m_height);
        GLdouble step=0.5;        //    角度
        GLfloat lw[2];
        glGetFloatv(GL_LINE_WIDTH_RANGE,lw);
        glLineWidth(lw[1]/8.0);
        glDisable(GL_LIGHTING);
        glBegin(GL_LINE_STRIP);
            glColor3f(0.5f,1.0f,0.2f);
            glVertex3d(m_bar1Length + m_radius,0.0,0.0);
            for(GLdouble i=0.0;i< = m_rotateAngle;i+ = step)
            {
                coord = DrawRelative1Point(m_height,
                    m_bar1Length,
                    m_radius,initLength,i);
                glVertex3d(coord.xCoord,coord.yCoord,0.0);
            }
        glEnd();
        glEnable(GL_LIGHTING);
}
```

```
//光照处理
void CJgfzView::EnableLighting()
{
        GLfloat light0Ambient[] = {0.1f,0.1f,0.1f,1.0f};
        GLfloat light0Diffuse[] = {0.8f,0.8f,0.8f,1.0f};
        GLfloat light0Position[] = {0.0f,1.0f,2.0f,1.0f};
        glLightfv(GL_LIGHT0,GL_AMBIENT,light0Ambient);
        glLightfv(GL_LIGHT0,GL_DIFFUSE,light0Diffuse);
        glLightfv(GL_LIGHT0,GL_POSITION,light0Position);
        glEnable(GL_LIGHTING);
        glEnable(GL_LIGHT0);
}

//时间机制
void CJgfzView::OnTimer(UINT nIDEvent)
{
        m_rotateAngle += m_speedFactor;
        if(m_rotateAngle > 360.0f)
        {
            m_rotateAngle = 0.0f;
            OnControlStatic();
        }
        Invalidate(FALSE);
        CView::OnTimer(nIDEvent);
}

//菜单控制
void CJgfzView::OnAbsolutemovement()
{
        KillTimer(1);
        m_rotateAngle = 0.0f;
        m_absoluteRelative = AbsoluteMovement;
        Invalidate(FALSE);
}

void CJgfzView::OnControlMove()
{
        if(m_MoveStatic == Move) return;
        m_MoveStatic = Move;
        m_radius = 0.0185f;
        SetTimer(1,50,0);
        Invalidate(FALSE);
}
```

```
void CJgfzView::OnControlStatic()
{
        if(m_MoveStatic==Static)        return;
        m_MoveStatic=Static;
        KillTimer(1);
        Invalidate(FALSE);
}

void CJgfzView::OnRelativemove1()
{
        m_rotateAngle=0.0f;
        m_absoluteRelative=RelativeMovement1;
        KillTimer(1);
        Invalidate(FALSE);
}

void CJgfzView::OnTrace1()
{
        m_absoluteRelative=RelativeTrace1;
        m_rotateAngle=0.0f;
        KillTimer(1);
        Invalidate(FALSE);
}

void CJgfzView::OnSpeedFast()
{
        m_speedFactor=25.0f;
        Invalidate(FALSE);
}

void CJgfzView::OnSpeedSlow()
{
        m_speedFactor=10.0f;
        Invalidate(FALSE);
}

void CJgfzView::OnSpeedUltra()
{
        m_speedFactor=35.0f;
        Invalidate(FALSE);
}
```

my.h 文件：

```
////////////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////                                  ////////////////////////////
////////////////////////////          my.h                    ////////////////////////////
////////////////////////////                                  ////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////////

//      my.h:

# include "gl \ glaux.h"

# define PI      3.14159265358979323846
# define RADIANS(x)        (x) * PI/180.0

typedef struct vector
{
        GLdouble xCoord,yCoord,zCoord;
}VERTEXCOORDINATE;

typedef struct lengthangle
{
        GLdouble bar2RotateAngle, taotongTranslate;
}LENGTHANGLEINFO;

//绘制圆环面:这是一个通用的绘制函数,其计算方法就是按平面曲线绕轴旋转而得
void torus(GLdouble innerRadius,GLdouble OuterRadius,int numc, int numt)
{
        int i, j, k;
        double s, t, x, y, z, xn,yn,zn,twopi,temp;
        double innerR,outerR;
        innerR = innerRadius;
        outerR = OuterRadius;
        if(outerR < innerR)
        {
            temp = outerR;
            outerR = innerR;
            innerR = temp;
        }
        twopi = 2 * PI;
        for (i = 0; i < numc; i++)
        {
            glBegin(GL_QUAD_STRIP);
                for (j = 0; j <= numt; i++)
```

```
            {
                for (k = 1; k >= 0; k--)
                {
                    s = (i + k) % numc + 0.5;
                    t = j % numt;

                    xn = cos(s * twopi/numc) * cos(t * twopi/numt);
                    yn = cos(s * twopi/numc) * sin(t * twopi/numt);
                    zn = sin(s * twopi / numc);

                    x = (outerR + innerR * cos(s * twopi/numc)) * cos(t * twopi/numt);
                    y = (outerR + innerR * cos(s * twopi/numc)) * sin(t * twopi/numt);
                    z = innerR * sin(s * twopi / numc);
                    glNormal3d(xn,yn,zn);
                    glVertex3d(x, y, z);
                }
            }
        glEnd();
    }
}

//    画杆:显示列表
void DrawBar(UINT nLists, GLfloat m_radius, GLfloat m_length)
{
        GLUquadricObj * quadObj;
        glNewList(nLists, GL_COMPILE);
            quadObj = gluNewQuadric();
            gluQuadricDrawStyle(quadObj, GLU_FILL);
            gluQuadricOrientation(quadObj, GLU_INSIDE);
            gluQuadricNormals(quadObj, GLU_SMOOTH);
            switch(nLists)
            {
            case 1: case 5:
                gluCylinder(quadObj, m_radius, m_radius, m_length, 10, 1);
                break;
            case 2:
                gluCylinder(quadObj, m_radius, m_radius, m_length, 8, 1);
                break;
            case 3:
                gluCylinder(quadObj, m_radius, m_radius, m_length, 15, 1);
                break;
            case 6: case 7: case 8:
                gluCylinder(quadObj, m_radius, m_radius, m_length, 15, 1);
                gluQuadricOrientation(quadObj, GLU_OUTSIDE);
```

```
                glPushMatrix();
                gluDisk(quadObj,0.0,m_radius,15,1);
                glTranslatef(0.0f,0.0f,m_length);
                gluDisk(quadObj,0.0,m_radius,15,1);
                glPopMatrix();
                break;
            default:    break;      // do nothing at all.
            }
        glEndList();
        gluDeleteQuadric(quadObj);
}

//    画套钩:
void DrawTaogua(GLfloat m_barlLength,GLfloat m_radius)
{
    GLfloat materialAD[] = {1.0f,0.0f,0.0f,1.0f};
    GLfloat materialSP[] = {0.90f,0.9f,0.9f,1.0f};
    glMaterialfv(GL_FRONT,GL_AMBIENT_AND_DIFFUSE,materialAD);
    glMaterialfv(GL_FRONT,GL_SPECULAR,materialSP);
    glMaterialf(GL_FRONT,GL_SHININESS,100.0);
    glPushMatrix();
        glTranslatef(0.0f,0.0f,m_barlLength + m_radius);
        glRotatef(92.0f,0.0f,1.0f,0.0f);
        torus((m_radius/2.0,m_radius,32,32);
    glPopMatrix();
}

//    返回第二杆旋转的角度及套筒需要移动的大小:参考函数 DrawRelative1Point 的说明
LENGTHANGLEINFO ClacuAngleBar2(GLdouble barlLength,
        GLdouble radius,GLdouble height,GLdouble rotateAngle)
{
    LENGTHANGLEINFO lengthAngle;
    GLdouble rotateRad;
    GLdouble yAxis;
    GLdouble xAxis;
    GLdouble angle;
    GLdouble length;
    /*    由于运动的特点以及 OpenGL 的机制,须对其进行分段处理,因为不同的角度段其计算的方法略有不同,在理解时,请将杆转动一个角度,然后用一直线画出第 2 杆,由三角几何关系获得计算机公司 */
    if(rotateAngle <= 90.0)
    {
        rotateRad = RADIANS(rotateAngle);
        yAxis = (barlLength + radius) * sin(rotateRad) + height;
        xAxis = (barlLength + radius) * cos(rotateRad);
```

```
        angle = atan2(yAxis,xAxis);
        angle = angle * 180.0/PI;
        length = sqrt(yAxis * yAxis + xAxis * xAxis);
        //      杆 2 旋转的角度
        lengthAngle.bar2RotateAngle = angle;
        //      套筒移动的大小
        lengthAngle.taotongTranslate = length;
    }
    else if(rotateAngle < = 180.0)
    {
        GLdouble minusAngle;
        minusAngle = 180.0 - rotateAngle;
        rotateRad = RADIANS(minusAngle);
        yAxis = (bar1Length + radius) * sin(rotateRad) + height;
        xAxis = (bar1Length + radius) * cos(rotateRad);
        angle = atan2(yAxis,xAxis);
        angle = angle * 180.0/PI;
        length = sqrt(yAxis * yAxis + xAxis * xAxis);
        lengthAngle.bar2RotateAngle = 180.0 - angle;
        lengthAngle.taotongTranslate = length;
    }
    else if(rotateAngle < = 270.0)
    {
        GLdouble minusAngle;
        minusAngle = 270.0 - rotateAngle;
        rotateRad = RADIANS(minusAngle);
        yAxis = height - (bar1Length + radius) * cos(rotateRad);
        xAxis = (bar1Length + radius) * sin(rotateRad);
        angle = atan2(yAxis,xAxis);
        angle = angle * 180.0/PI;
        length = sqrt(yAxis * yAxis + xAxis * xAxis);
        lengthAngle.bar2RotateAngle = 180.0 - angle;
        lengthAngle.taotongTranslate = length;
    }
    else if(rotateAngle < = 360.0)
    {
        GLdouble minusAngle;
        minusAngle = 360.0 - rotateAngle;
        rotateRad = RADIANS(minusAngle);
        yAxis = height - (bar1Length + radius) * sin(rotateRad);
        xAxis = (bar1Length + radius) * cos(rotateRad);
        angle = atan2(yAxis,xAxis);
        angle = angle * 180.0/PI;
        length = sqrt(yAxis * yAxis + xAxis * xAxis);
```

```
            lengthAngle.bar2RotateAngle = angle;
            lengthAngle.taotongTranslate = length;
        }
        return lengthAngle;
}

//实时地绘制绝对轨迹:
void DrawCircleRoad(GLdouble bar1Length,GLdouble radius,
                    GLdouble rotateAngle,GLfloat sign)
{
        //    参数 rotateAngle 是当前旋转的角度
        //    杆 1 的长度加上挂钩的半径才是曲柄的真正长度:
        GLdouble step = 0.01;
        GLdouble length = bar1Length + radius;
        GLfloat lw[2];
        GLdouble rotateRad = RADIANS(rotateAngle);
        glGetFloatv(GL_LINE_WIDTH_RANGE,lw);
        glLineWidth(lw[1]/8.0);
        glDisable(GL_LIGHTING);
        glColor3f(1.0f,1.0f,0.0f);
        glBegin(GL_LINE_STRIP);
        for(GLdouble i = 0.0;i <= rotateRad;i += step)
                glVertex3f(0.0f,sign * length * sin(i),sign * length * cos(i));
        glEnd();
        glEnable(GL_LIGHTING);
}

/*绘制下支座:下支座是由一条位于 o_xy 平面上的曲线沿 z 轴拉抻的结果,这条弯曲的曲线分为两段,一段为一条抛物线,其方程为 y = c * x * x,其中,系数 c 为 zizuoCoefficient,x 的取值范围为[ - zizuoWidth1/2.0,zizuoWidth1/2.0]。另一段为两条直线段,分别由抛物线两端点连接点(zizuoWidth2,0)和(-zizuoWidth2,0)构成。拉抻的厚度为 zizuoThick。事实上,程序制作了两片,把已制作的一片沿 z 轴平移 zizuoOffset,得到另一片,最后将这两片的底部由两个四边形来连接,它是一个等腰梯形。radius 为支座销钉(为一个圆柱)的半径
*/
void DrawZiZuoXia(GLdouble zizuoCoefficient,
            GLdouble zizuoThick,GLdouble zizuoWidth1,
            GLdouble zizuoWidth2,GLdouble zizuoOffset,
            GLdouble radius)
{
        GLfloat materialAD[] = {0.5f,0.50f,0.50f,1.0f};
        GLfloat materialSP[] = {0.90f,0.9f,0.9f,1.0f};
        glMaterialfv(GL_FRONT_AND_BACK,GL_AMBIENT_AND_DIFFUSE,materialAD);
        glMaterialfv(GL_FRONT_AND_BACK,GL_SPECULAR,materialSP);
        glMaterialf(GL_FRONT_AND_BACK,GL_SHININESS,100.0);
```

```
glPolygonMode(GL_FRONT_AND_BACK,GL_FILL);
glFrontFace(GL_CCW);
GLdouble xAxis = zizuoWidth1/2.0;
GLdouble yAxis = -zizuoCoefficient * xAxis * xAxis;
double i;
glPushMatrix();
glTranslatef(0.0f,0.0f,zizuoThick - 0.05f);
glBegin(GL_POLYGON);
    glNormal3f(0.0f,0.0f,1.0f);
    for(i = xAxis;i >= -xAxis;i -= 0.001)
    {
        yAxis = -zizuoCoefficient * i * i;
        glVertex3d(i,yAxis,0.0f);
    }
    yAxis = -zizuoCoefficient * xAxis * xAxis;
    glVertex3d(-xAxis,yAxis,0.0f);
glEnd();
glPopMatrix();
GLfloat materialAD2[] = {1.0f,1.0f,1.0f,1.0f};
GLfloat materialSP2[] = {0.90f,0.9f,0.9f,1.0f};
glMaterialfv(GL_FRONT_AND_BACK,GL_AMBIENT_AND_DIFFUSE,materialAD2);
glMaterialfv(GL_FRONT_AND_BACK,GL_SPECULAR,materialSP2);
glMaterialf(GL_FRONT_AND_BACK,GL_SHININESS,100.0);
//    第一部分曲面:(简化,没有给出曲面精确法向量)
glBegin(GL_QUAD_STRIP);
    for(i = xAxis;i >= -xAxis;i -= 0.001)
    {
        yAxis = -zizuoCoefficient * i * i;
        glVertex3d(i,yAxis,0.0f);
        glVertex3d(i,yAxis,zizuoThick);
    }
glEnd();
GLdouble zBack = zizuoThick + 2 * radius + zizuoOffset;
GLdouble zFront = zBack + zizuoThick;
//    第二部分曲面:
glBegin(GL_QUAD_STRIP);
    for(i = xAxis;i >= -xAxis;i -= 0.001)
    {
        yAxis = -zizuoCoefficient * i * i;
        glVertex3d(i,yAxis,zBack);
        glVertex3d(i,yAxis,zFront);
    }
glEnd();
glMaterialfv(GL_FRONT_AND_BACK,GL_AMBIENT_AND_DIFFUSE,materialAD);
```

```
glMaterialfv(GL_FRONT_AND_BACK,GL_SPECULAR,materialSP);
glMaterialf(GL_FRONT_AND_BACK,GL_SHININESS,100.0);
//    第二部分后表面:
glBegin(GL_POLYGON);
    glNormal3f(0.0f,0.0f,-1.0f);
    for(i=xAxis;i>=-xAxis;i-=0.001)
    {
        yAxis=-zizuoCoefficient*i*i;
        glVertex3d(i,yAxis,zBack);
    }
    yAxis=-zizuoCoefficient*xAxis*xAxis;
    glVertex3d(-xAxis,yAxis,zBack);
glEnd();
//    第二部分前表面:
glBegin(GL_POLYGON);
    glNormal3f(0.0f,0.0f,1.0f);
    for(i=xAxis;i>=-xAxis;i-=0.001)
    {
        yAxis=-zizuoCoefficient*i*i;
        glVertex3d(i,yAxis,zBack+zizuoThick);
    }
    yAxis=-zizuoCoefficient*xAxis*xAxis;
    glVertex3d(-xAxis,yAxis,zBack+zizuoThick);
glEnd();
//    计算结合部位的切线,以使表面光滑。计算节点坐标:
GLdouble tangent=zizuoCoefficient*zizuoWidth1;
GLdouble yAxis2=-zizuoCoefficient*xAxis*xAxis;
yAxis=yAxis2-(zizuoWidth2-zizuoWidth1)*tangent;
//    台体后表面:
glBegin(GL_POLYGON);
    glVertex3d(zizuoWidth2/2.0,yAxis,0.0f);
    glVertex3d(zizuoWidth1/2.0,yAxis2,0.0f);
    glVertex3d(-zizuoWidth1/2.0,yAxis2,0.0f);
    glVertex3d(-zizuoWidth2/2.0,yAxis,0.0f);
glEnd();
//    台体前表面:
glBegin(GL_POLYGON);
    glVertex3d(zizuoWidth2/2.0,yAxis,zFront);
    glVertex3d(zizuoWidth1/2.0,yAxis2,zFront);
    glVertex3d(-zizuoWidth1/2.0,yAxis2,zFront);
    glVertex3d(-zizuoWidth2/2.0,yAxis,zFront);
glEnd();
//    台体侧翼及上下表面:
glMaterialfv(GL_FRONT_AND_BACK,GL_AMBIENT_AND_DIFFUSE,materialAD2);
```

```
        glMaterialfv(GL_FRONT_AND_BACK,GL_SPECULAR,materialSP2);
        glMaterialf(GL_FRONT_AND_BACK,GL_SHININESS,100.0);
        glBegin(GL_QUAD_STRIP);
            glVertex3d(zizuoWidth2/2.0,yAxis,0.0f);
            glVertex3d(zizuoWidth2/2.0,yAxis,zFront);
            glVertex3d(zizuoWidth1/2.0,yAxis2,0.0f);
            glVertex3d(zizuoWidth1/2.0,yAxis2,zFront);
            glVertex3d(-zizuoWidth1/2.0,yAxis2,0.0f);
            glVertex3d(-zizuoWidth1/2.0,yAxis2,zFront);
            glVertex3d(-zizuoWidth2/2.0,yAxis,0.0f);
            glVertex3d(-zizuoWidth2/2.0,yAxis,zFront);
        glEnd();
}

//绘制一个填充圆,也可以用实用库函数来绘制
void CircleFill(GLdouble radius)
{
        glPolygonMode(GL_FRONT_AND_BACK,GL_FILL);
        glFrontFace(GL_CCW);
        double degree=0.0;
        double radians=0.0;
        glBegin(GL_POLYGON);
            glVertex3d(radius,0.0,0.0);
            for(int i=1;i<=36;i++)
            {
                degree+=10.0;
                radians=RADIANS(degree);
                glVertex3d(radius*cos(radians),radius*sin(radians),0.0);
            }
        glEnd();
}

/*获取动点位置:动点即为在初始位置杆2上与OA杆上的A点相重合的点,动点在运动的过程中始终是这个固定不动的点。
由于几何位图关系,因此要分成4个阶段,0-90,90-180,180-270,270-360。
OA杆是主动杆,其位置由它与x轴的转角来唯一决定,这样可以决定套筒M和O1B杆的位置:套筒M由平移和转动来决定,而O1B杆则由一个转动角来决定其位置。*/
VERTEXCOORDINATE DrawRelative1Point(GLdouble height,GLdouble bar1Length,
                    GLdouble radius,GLdouble initLength,GLdouble rotateAngle)
{
    //    变量rotateAngle以角度表示
    GLdouble mnLength;
    GLdouble omLength;
    GLdouble radiusConst;
```

```
GLdouble cosAngle;
GLdouble sinAngle;
VERTEXCOORDINATE coord;
LENGTHANGLEINFO lengthAngle;
radiusConst = bar1Length + radius;
//    获得摆杆(杆 2)的运动信息:
lengthAngle = ClacuAngleBar2(bar1Length, radius, height, rotateAngle);
GLdouble rotateBar1 = RADIANS(rotateAngle);
GLdouble rotateBar2 = RADIANS(lengthAngle.bar2RotateAngle);
//    下面对当前角度进行判断:
if(rotateBar1 < = PI/2.0)
{
    mnLength = lengthAngle.taotongTranslate - initLength;
    omLength = height * height + initLength * initLength
        - 2.0 * height * initLength * cos(3.14158265/2.0 - rotateBar2);
    omLength = sqrt(omLength);
    cosAngle = (radiusConst * radiusConst + omLength * omLength
        - mnLength * mnLength)/(2 * radiusConst * omLength);
    sinAngle = 1.0 - cosAngle * cosAngle;
    sinAngle = sqrt(sinAngle);
    coord.xCoord = omLength * cosAngle;
    coord.yCoord = - omLength * sinAngle;
}
else if(rotateBar1 < = PI)
{
    mnLength = lengthAngle.taotongTranslate - initLength;
    omLength = height * height + initLength * initLength
        - 2.0 * height * initLength * cos(rotateBar2 - PI/2.0);
    omLength = sqrt(omLength);
    cosAngle = (radiusConst * radiusConst + omLength * omLength
        - mnLength * mnLength)/(2 * radiusConst * omLength);
    sinAngle = 1.0 - cosAngle * cosAngle;
    sinAngle = sqrt(sinAngle);
    coord.xCoord = omLength * cosAngle;
    coord.yCoord = omLength * sinAngle;
}
else if(rotateBar1 < = PI * 1.5)
{
    mnLength = initLength - lengthAngle.taotongTranslate;
    omLength = height * height + initLength * initLength
        - 2.0 * height * initLength * cos(rotateBar2 - PI/2.0);
    omLength = sqrt(omLength);
    cosAngle = (radiusConst * radiusConst + omLength * omLength
```

```
            - mnLength * mnLength)/(2 * radiusConst * omLength);
        sinAngle = 1.0 - cosAngle * cosAngle;
        sinAngle = sqrt(sinAngle);
        coord.xCoord = omLength * cosAngle;
        coord.yCoord = - omLength * sinAngle;
    }
    else
    {
        mnLength = initLength - lengthAngle.taotongTranslate;
        omLength = height * height + initLength * initLength
            - 2.0 * height * initLength * cos(PI/2.0 - rotateBar2);
        omLength = sqrt(omLength);
        cosAngle = (radiusConst * radiusConst + omLength * omLength
            - mnLength * mnLength)/(2 * radiusConst * omLength);
        sinAngle = 1.0 - cosAngle * cosAngle;
        sinAngle = sqrt(sinAngle);
        coord.xCoord = omLength * cosAngle;
        coord.yCoord = omLength * sinAngle;
    }
    coord.zCoord = 0.0;
    return coord;
}
```

附录　配书 CD-ROM 内容介绍

下面详细说明本书附带的 CD-ROM 中的内容和使用方法。

(1) exe 目录

该目录中包括本书中 18 个工程文件所生成的可执行文件。可以事先运行它们,看看运行结果(请用 VC 6.0)。但前提是你的 Windows 95(98)的 system 目录中包含有 glu32.dll 和 opengl32.dll 两个库文件。一般来说 Windows 98 已自动带有这两个库。

(2) Programs 目录

该目录中包括本书的所有源程序,它们可在 VC 5.0 和 VC 6.0 下运行。一个目录一个工程文件集,因而该目录下共有 18 个子目录。比如,第 3 章,在书中读者会根据指示建立一个工程名为 StepinGl 的工程文件,同时在此光盘的 Programs 目录中就有一个对应的目录,其名就是 StepinGl。其他以此类推。作者是在 Windows 98 和 VC 5.0 上开发的。但是作者已经在 VC 6.0 上完全测试过,没有任何问题! 注意,本书的所有程序都是在 MFC 环境中生成的。

使用方法:

第一,将某个要运行或要学习的工程文件目录(比如,读者在学习第 3 章时,就是 StepinGl 这个目录)拷贝至硬盘。

第二,在资源管理器中将目录下的所有文件的属性改为可读可写,即取消文件的只读属性。

第三,启动 VC 5.0 或 VC 6.0。

第四,执行 File \ Open Workspace… \ ,在通用文件打开对话框中将目录切换到刚才从光盘拷贝文件至硬盘的目的目录(比如:硬盘上的 StepinGl 目录),选择扩展名为 dsw 的文件,双击它就打开了这个工程文件的开发环境。

第五,确保你的 Windows 95(98)的 system 目录下有 opengl32.dll 和 glu32.dll 库文件。

第六,请按 ctrl + F5 键,编译连接并运行程序。

(3) result _ bmp 目录

该目录下有本书的程序所生成的部分结果,作者把它们抓成位图的形式,以供读者参考。共 76 幅图片。

(4) win32 _ GL 目录

这是作者专门为本书光盘编制的一个可在 Win32 Application 开发环境下运行的 OpenGL 程序。它的代码并不多,相当于一个模板,其作用相当于书中第 3 章的极小化 OpenGL 应用程序。它的好处是不需要 MFC 的库文件,因而应用程序较轻便。在没有 MFC 库的环境下,只要系统有 opengl32.dll 和 glu32.dll 两个库文件就可以运行程序所生成的可执行文件。

使用方法:

第一,启动 VC 5.0 或 VC 6.0。

第二,执行 Flie \ new... \ 菜单,在弹出的对话框中选择 Projects 标签,在其列表框中选择 Win32 Application,在右边的 Projects 标签,在其列表框中选择 Win32 Application,在右边的 Project Name 栏中输入工程文件名,比如:OK 按钮。在弹出的对话框中确保选择了 An empty project 单选项,单击 Finish 按钮和 OK 按钮。这样就生成了一个 Win32 Application 应用程序的开发环境。

第三,从光盘的 win32 _ GL \ SpinLine \ 目录下拷贝 SpinLine.cpp 文件至你刚建立的工程文件所在的目录,若按作者刚才的步骤,这个目录应为 e: \ Win32AppOpenGL \ 。

第四,执行菜单 Project \ Add To Project \ File... \ ,在弹出的对话框中选择 SpinLine.cpp 文件。这样就将该文件添加进了该工程中。

第五,在左边的 Workspace 窗口中选择 FileView 标签,确保单击了 Win32AppOpenGlfiles 字样。

第六,执行菜单 Project \ Setting... \ ,在弹出的对话框中选择 Link 标签,在 Object/libaray modules:栏的尾部键入:opengl32.lib glu32.lib glu32.lib(每个文件请用空格隔开)。

第七,请按 Ctrl + F5 键,编译连接并运行程序。

Document generated by Anna’s Archive around 2023-2024 as part of the DuXiu collection (https://annas-blog.org/duxiu-exclusive.html).

Images have been losslessly embedded. Information about the original file can be found in PDF attachments. Some stats (more in the PDF attachments):

{
  "filename": "MTAwMjg1NzEuemlw",
  "filename_decoded": "10028571.zip",
  "filesize": 25039550,
  "md5": "34f1badad323eb4a53d8c1b88453f5df",
  "header_md5": "18ff587d186b01b9802a3e28cfe99432",
  "sha1": "f544a29ed233dc031e0297d4953319a4f4acc960",
  "sha256": "a3d55bda9d55705e68d39fac501fd94a8183c89e10bb3c70eb6e6552c833e4e4",
  "crc32": 851347309,
  "zip_password": "",
  "uncompressed_size": 25742993,
  "pdg_dir_name": "OpenGL\u2592\u03b1\u2502\u2560\u2559\u03b4\u2569\u2561\u2514\u00b2_10028571",
  "pdg_main_pages_found": 471,
  "pdg_main_pages_max": 471,
  "total_pages": 484,
  "total_pixels": 3444126562,
  "pdf_generation_missing_pages": false
}